UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

DISEÑO Y DESARROLLO DE UNA EXTENSIÓN DE NEO4J PARA LA GESTIÓN DE CADENAS DE ADN

Ana Peraita Alonso

Tutor: Simone Santini

Junio 2017

DISEÑO Y DESARROLLO DE UNA EXTENSIÓN DE NEO4J PARA LA GESTIÓN DE CADENAS DE ADN

AUTOR: Ana Peraita Alonso

TUTOR: Simone Santini

Dpto. de Ingeniería Informática

Escuela Politécnica Superior

Universidad Autónoma de Madrid

Junio de 2017

Resumen (castellano)

Desde hace un tiempo, dadas las recientes investigaciones, se han comenzado a generar una gran cantidad de datos biológicos que requieren ser tratados y guardados. Existen varias aplicaciones que trabajan con este tipo de datos, pero no un ecosistema que implemente operaciones con estos datos y a partir del cual se puedan crear las aplicaciones que se necesiten.

El objetivo de este Trabajo de Fin de Grado es la construcción de varias operaciones individuales que trabajen con datos biológicos, como son cadenas de ADN y proteínas, sobre un sistema que soporte guardar tanto estos datos biológicos ya nombrados como los resultados de las distintas operaciones individuales. Para ello hemos decidido usar una base de datos basada en grafos Neo4j y sobre esta base de datos implementar extensiones a los algoritmos de Smith Waterman, para el alineamiento de cadenas, el algoritmo de síntesis de proteínas y el algoritmo probabilístico basado en el algoritmo de Chou Fasman, para la predicción de la estructura secundaria de las proteínas.

Además se ha probado la eficacia de estas implementaciones ejecutando varias peticiones sobre la base de datos Neo4j y midiendo los tiempos así como obteniendo el esquema de ejecución de varias peticiones. Gracias a estas pruebas hemos concluido que el tiempo que tardan las peticiones que ejecutan los tres procedimientos es razonable y que, las búsquedas en la base de datos sólo serán eficientes si no se realizan peticiones de todos los datos, por lo tanto en las peticiones de búsqueda que se realicen desde otras aplicaciones se deberán filtrar los nodos.

Finalmente se ha desarrollado una aplicación de prueba que realiza llamadas a los distintos procedimientos de la base de datos, para después enseñarlos a modo de demo de la potencia del producto creado.

Abstract (English)

Lately, due to some investigations, a great quantity of biological data is being generated, and all of that data requires being treated and stocked properly. There are several applications that work with this kind of data but there is not an ecosystem that implements operations with this kind of data and from which all the applications needed can be created.

The goal of this Bachelor Thesis is to build several individual operations that work with biological data, as DNA and protein sequences, on a system that endures saving all the biological data that we named before and the results of the different individual transactions. So we decided to use a graph database Neo4j and to add to the database the functionality of several extensions such as the Smith Waterman algorithm, for the sequence alignment, the protein synthesis algorithm and the probabilistic algorithm based on the Chou Fasman algorithm, for the protein second structure prediction.

Furthermore, we have tested the effectiveness of these implementations by executing several queries on the Neo4j database and measuring the times and obtaining the execution plan of several requests. Thanks to those tests, we have concluded that the time the requests delays to execute the three procedures is reasonable and that the searches on the database will only be efficient if we don't ask for a huge amount of data, therefore on the search queries that are made from other applications some nodes should be filtered.

Finally, we have developed a test application that makes calls to the different procedures of the database, and then show them as a demo of the power of the created product.

Palabras clave (castellano)

En este apartado se incluyen una serie de palabras clave con el fin de indexar el proyecto.

- ADN
- Proteínas
- Genética
- Biología
- Bioinformática
- Neo4j

Keywords (inglés)

A set of keywords are included in this section for the sake of indexing the project.

- DNA
- Proteins
- Genetics
- Biology
- Bioinformática
- Neo4j

Agradecimientos

Me gustaría aprovechar este espacio para dedicarles unas palabras a todas aquellas personas que aun sin saberlo, han hecho este trabajo posible.

En primer lugar me gustaría agradecer a mi tutor, Simone Santini, su apoyo durante la realización de este trabajo y la oportunidad que me ha dado de llevar a cabo este proyecto a su lado.

Por supuesto quiero agradecer también a mis amigos y familiares puesto que ellos han sido quienes más me han ayudado a llegar hasta aquí y me han animado y apoyado en todo momento. Muchas gracias por estar ahí.

Y por último, a todos los demás que, me han ayudado a llegar a este momento, gracias de todo corazón.

INDICE DE CONTENIDOS

INDICE DE ILUSTRACIONES

INDICE DE ESQUEMAS DE EJECUCIÓN

Introducción

1.1 Motivación

Actualmente en el mundo de la biología, y en el de la ciencia en general, uno de los mayores problemas existentes es la gran cantidad de información disponible y por tanto la dificultad intrínseca de manejar dicha cantidad de datos con el fin de poder analizarlos de manera global y establecer relaciones entre ellos. Por este motivo la bioinformática, aplicación de las tecnologías de la información a la gestión y análisis de información biológica, ha sufrido un gran crecimiento durante los últimos años desarrollando diversas soluciones a varios de los problemas que se presentan ante la comunidad científica.

Un claro ejemplo de este problema es el tratamiento de información de carácter biológico como son las proteínas y la información genética. Existen grandes bases de datos en las cuales podemos encontrar grandes cantidades de datos de carácter biológico, entre las que destacan GenBank [1], European Nucleotide Archive [2], UniProt [3] y Protein Data Bank [4] .Todas estas bases de datos proveen de una gran cantidad de datos a obtener y con los que trabajar, pero todas ellas ofrecen sus servicios como una aplicación, no existe un ecosistema de desarrollo de aplicaciones para realizar operaciones definidas con datos de tipo biológico. Por lo tanto, si se desea trabajar con datos de este tipo en una aplicación se debe diseñar e implementar una aplicación completa con los distintos algoritmos que se desean ejecutar.

Además, las bases de datos que se han destacado en el párrafo anterior, principalmente que realizan es la operación de alineamiento, ofrecen muy poca funcionalidad extra como pueden ser algoritmos de síntesis proteica o de predicción de los distintos niveles estructurales de las proteínas.

1.2 Objetivos

El principal objetivo de este Trabajo de Fin de Grado es rellenar este hueco en el mercado del que he hablado en el apartado anterior y por lo tanto crear un ecosistema que permita trabajar con distintos datos biológicos y realizar diversas operaciones con ellos.

Para realizar este proyecto se necesita una base de datos y se consideró por motivos que se explican en el apartado 3.1 elegir Neo4j, una base de datos basada en grafos y que permite implementar extensiones a su funcionalidad.

Puesto que hay muchos distintos tipos de operaciones entre datos biológicos, se decidió elegir entre ellas 3 como una prueba de concepto del ecosistema que queremos crear: la alineación genética, la síntesis de proteínas y la predicción de la estructura secundaria de las proteínas. Estas tres operaciones han sido elegidas porque todas ellas son lo suficientemente complejas como para justificar la creación del ecosistema mencionado a su alrededor y significativas respecto al problema formulado. Por lo tanto uno de los objetivos de este proyecto es crear tres extensiones para la base de datos Neo4j, una por cada uno de los algoritmos.

Puesto que para trabajar con este tipo de datos es obligatorio que el sistema sea eficiente, el segundo objetivo de este proyecto será evaluar la eficiencia de dichas extensiones en un entorno de desarrollo controlado y varias de las llamadas más comunes que se pueden realizar hacia la base de datos.

Finalmente, es necesario presentar el sistema de alguna manera para acercar a posibles usuarios la potencia de este ecosistema, por lo tanto el tercer objetivo es crear una aplicación que permita trabajar con las tres extensiones creadas.

1.3 Organización de la memoria

En el siguiente capítulo se realizara una exposición del Estado del arte, que ofrecerá una visión de los trabajos realizados previamente y así poderlos relacionar con el tratado en este documento.
En el capítulo 3 se realizará un análisis de las distintas decisiones de diseño que se han tomado para realizar la aplicación.
En el capítulo 4 consiste en una explicación general de las distintas.
En el capítulo 5 se explicará cómo se han diseñado y desarrollado tanto las extensiones como la aplicación de prueba
En el capítulo 6 tenemos distintas pruebas para testear la eficiencia de las extensiones de la base de datos y del diseño del modelo de datos asociado.
Finalmente, en el capítulo 7 se verán las conclusiones del trabajo realizado, así como las aportaciones futuras que se pueden añadir al trabajo presentado.

2 Estado del arte

Esta sección introduce de forma general el contexto de este trabajo. Aquí se puede leer una explicación sobre los tres tipos de algoritmos utilizados en este trabajo (el algoritmo de alineación, el algoritmo de síntesis de proteínas y el algoritmo de previsión de la estructura secundaria de las proteínas) y de los problemas que solucionan. Posteriormente se encontrará una comparativa de distintos tipos de bases de datos, para luego centrarnos en la base de datos basada en grafos Neo4j.

2.1 Cadenas de ADN: Información básica de su estructura

El ácido desoxirribonucleico (ADN) compone el material genético de todos los organismos celulares y casi todos los virus. En él se guarda toda la información necesaria para controlar el metabolismo de un ser vivo [5].

Estructura del ADN

Cada molécula de ADN está constituida por dos cadenas o bandas formadas por un elevado número de compuestos químicos llamados nucleótidos. Estas cadenas forman una doble hélice. Cada nucleótido está formado por tres unidades: una molécula de azúcar llamada desoxirribosa, un grupo fosfato y uno de cuatro posibles compuestos nitrogenados llamados bases: adenina (abreviada como A), guanina (G), timina (T) y citosina (C) [6].

Los nucleótidos de cada una de las dos cadenas que forman el ADN establecen una asociación específica con los correspondientes de la otra cadena. Debido a la afinidad química entre las bases, los nucleótidos que contienen adenina se acoplan siempre con los que contienen timina, y los que contienen citosina con los que contienen guanina. Las bases complementarias se unen entre sí por enlaces químicos débiles llamados enlaces de hidrógeno. Durante la mitosis celular las cadenas complementarias de ADN se separan para duplicarse y generar el material genético de la nueva célula. Además, las cadenas de ADN también se separan para realizar la traducción a ARN, la cual explicamos con más detalle en el apartado de traducción [5].

2.2 El problema de alineación genética

El alineamiento de secuencias es un campo muy importante en bioinformática [7] y tiene un gran número de aplicaciones como por ejemplo predicción de la funcionalidad (encontrando genes similares), encontrar ORFs (Open Reading Frames: Partes del código genético que pueden ser traducidos), identificación de genes homólogos [8] [9].

Tipos de alineamiento:

Un alineamiento de secuencias es una forma de representar y comparar dos o más cadenas para resaltar sus zonas de similitud, en este caso usaremos secuencias biológicas como son cadenas de ADN o proteínas [10].

Los tipos de alineamiento se dividen según el alcance en:

- Alineamiento global: Alineamiento de todos los pares contra todos los pares (el algoritmo más común es el de Needleman-Wunsch) [11]. Esta técnica permite obtener unos resultados muy interesantes cuando las cadenas que se analizan presentan un alto grado de similitud (en su longitud y contenido) [12] [13].
  Un ejemplo es:

```
aaagcggaagtcacag
||.||.||||| |.||
aaggctgaagt-atag
```

Como podemos ver en el ejemplo el alineamiento se produce en toda la cadena.

- Alineamiento local: Alineamiento de subsecuencias de pares (el algoritmo más común es el de Smith Waterman) [11] Este tipo de alineamiento es utilizado en cadenas con baja similitud pero que pueden contener regiones similares [12]. El alineamiento local es muy importante para comparar pequeñas zonas del genoma que se espera que sean parecidas, puesto que en caso de usar un algoritmo global se tardaría mucho.
  Por ejemplo tenemos:

```
aaagcggaagtcacag
......||||| ....
aaggctgaagt-atag
```

  Como podemos ver en este ejemplo el alineamiento está sólo en una parte de las cadenas alineadas.

Observando los ejemplos vemos cómo son muy distintos entre ellos, esto es porque cada uno de los algoritmos busca el alineamiento óptimo usando distintos criterios.

Podemos dividir los algoritmos en:

- Programas de alineamiento por pares
  - Programas de punto matriz:
    - Consiste en construir una matriz con ambas cadenas y colocar un punto en la casilla que coincida. Aquellas secuencias que coincidan aparecerán en la diagonal
  - Programas de programación dinámica
    - Los algoritmos más importantes son Smith Waterman y Nedleman Wunch
    - Principalmente consiste en, teniendo una matriz de scores y modificarlos usando de los parámetros de emparejamiento (positivo), no emparejamiento (negativo) y penalización por espacio entre emparejamientos.
  - Programas de palabras
    - No garantizan encontrar el alineamiento óptimo pero son bastante más eficientes que la programación dinámica
    - Encuentran una serie de pequeñas subsecuencias en las secuencias no superpuestas a testear.

2.3 El algoritmo de Smith Waterman

Estrategia para realizar el alineamiento local de secuencias biológicas. Este algoritmo pertenece al conjunto de algoritmos de programación dinámica lo cual implica que encuentra el alineamiento óptimo, aun así presenta un gran coste computacional O(m*n) siendo m y n las longitudes de las cadenas a alinear [12] [14] [15].

Es un algoritmo muy usado para alinear secuencias biológicas, de entre los programas que lo usan destaca el BLAST (**Basic Local Alignment Search Tool**) [16].

El proceso del algoritmo se puede separar en dos fases claramente diferenciadas:

- Generación de la matriz de resultados
  Se obtiene la matriz ejecutando sobre las cadenas:

$$H_{ij} = \max\left\{H_{i-1,j-1} + s(a_i, b_j)\,, \max_{k \geq 1}\{H_{i-k,j} - W_k\}, \max_{l \geq 1}\{H_{i,j-l} - W_l\}, 0\right\}$$

$1 \leq i \leq n\ y\ 1 \leq j \leq m$

$H_{k0} = H_{0t} = 0\ para\ 0 \leq k \leq n\ y\ 0 \leq l \leq m$

Siendo n y m las longitudes de las cadenas a analizar [17].

- Proceso de BackTrace desde el punto donde se encuentre el valor máximo de la matriz. Empezando en el mejor score que se da en la matriz anterior y acabando en una casilla que tenga 0, volver hacia la fuente de cada uno de los scores, esto obtendrá el mejor alineamiento local [18] [19] [20].

Pseudocódigo del algoritmo de Smith Waterman:

Input: two sequences X and Y
Output: optimal local alignment and score α
Initialization: Set $F(i,0) := 0$ for all $i = 0, 1, 2, \ldots, n$
Set $F(0,j) := 0$ for all $j = 1, 2, \ldots, m$
For $i = 1, 2, \ldots, n$ **do**:
  For $j = 1, 2, \ldots, m$ **do**:
    Set $F(i,j) := \max \begin{cases} 0 \\ F(i-1, j-1) + s(x_i, y_j) \\ F(i-1, j) - d \\ F(i, j-1) - d \end{cases}$
    Set backtrace $T(i,j)$ to the maximizing pair (i', j')
Set $(i,j) := \arg\max\{F(i,j) \mid i = 1, 2, \ldots, n, j = 1, 2, \ldots, m\}$
The best score is $\alpha := F(i,j)$
repeat
  if $T(i,j) = (i-1, j-1)$ **print** $\binom{x_{i-1}}{y_{j-1}}$
  else if $T(i,j) = (i-1, j)$ **print** $\binom{x_{i-1}}{-}$ **else print** $\binom{-}{y_{j-1}}$
  Set $(i,j) := T(i,j)$
until $F(i,j) = 0$.

2.4 Traducción ADN – Proteínas

En los seres vivos, para traducir de ADN a proteínas se lleva a cabo un proceso que consta de dos partes [5] [21]:

Transcripción: Consiste en pasar una secuencia ADN (dirección 3’ – 5’) a una secuencia ARN complementaria (dirección 5’ -3’). Las secuencias de ARN pueden ser de distintos tipos, pero de entre ellas destacaremos ARNm que es el ARN de tipo mensajero, que será transformado en proteínas.

Traducción: Consiste en la síntesis de las proteínas a partir del ARNm, en la cual cada codón (triplete de ADN) se traduce en un aminoácido a partir de la secuencia de inicio.

Aunque este es en líneas generales el proceso que se sigue para sintetizar proteínas en los organismos vivos, por propósitos biomédicos se ha obtenido una tabla de traducción de ADN en dirección 5’ – 3’ a proteínas [22]

Standard genetic code

1st base	2nd base								3rd base
	T		C		A		G		
T	TTT	(Phe/F) Phenylalanine	TCT	(Ser/S) Serine	TAT	(Tyr/Y) Tyrosine	TGT	(Cys/C) Cysteine	T
	TTC		TCC		TAC		TGC		C
	TTA	(Leu/L) Leucine	TCA		TAA[B]	Stop (*Ochre*)	TGA[B]	Stop (*Opal*)	A
	TTG		TCG		TAG[B]	Stop (*Amber*)	TGG	(Trp/W) Tryptophan	G
C	CTT		CCT	(Pro/P) Proline	CAT	(His/H) Histidine	CGT	(Arg/R) Arginine	T
	CTC		CCC		CAC		CGC		C
	CTA		CCA		CAA	(Gln/Q) Glutamine	CGA		A
	CTG		CCG		CAG		CGG		G
A	ATT	(Ile/I) Isoleucine	ACT	(Thr/T) Threonine	AAT	(Asn/N) Asparagine	AGT	(Ser/S) Serine	T
	ATC		ACC		AAC		AGC		C
	ATA		ACA		AAA	(Lys/K) Lysine	AGA	(Arg/R) Arginine	A
	ATG[A]	(Met/M) Methionine	ACG		AAG		AGG		G
G	GTT	(Val/V) Valine	GCT	(Ala/A) Alanine	GAT	(Asp/D) Aspartic acid	GGT	(Gly/G) Glycine	T
	GTC		GCC		GAC		GGC		C
	GTA		GCA		GAA	(Glu/E) Glutamic acid	GGA		A
	GTG		GCG		GAG		GGG		G

Ilustración 1 Tripletes asociados a aminoácidos

Esta tabla principalmente consiste en el triplete de ADN y la respectiva proteína a la que será traducido.

2.5 La estructura de las proteínas

Las proteínas son compuestos orgánicos que son el constituyente esencial de las células vivas. Están formadas por varios aminoácidos, unidos por un enlace peptídico. Existen un total de 22 aminoácidos que son codificados y sintetizados por algún ser vivo a partir de la información contenida en el ADN y transcrita al ARN. Los distintos aminoácidos crean enlaces débiles entre ellos, por lo tanto la composición de una proteína influye de manera determinante en su forma y, a la vez su forma determina en parte su función [5] [21].

La estructura de las proteínas se ha clasificado según sus niveles de organización como podemos ver en la siguiente imagen [23].

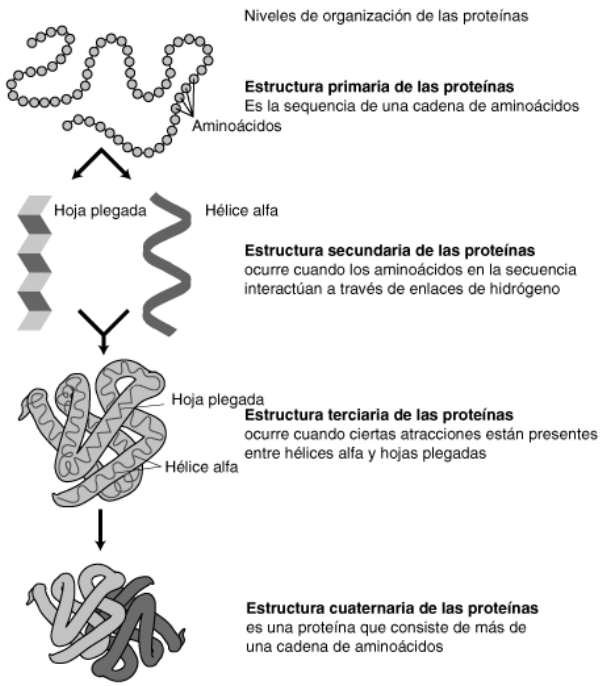


Ilustración 2 Niveles de organización de las proteínas

Para este trabajo nos centraremos en los tipos de estructura secundaria:

La estructura secundaria es el plegamiento que la cadena polipeptídica adopta gracias a la formación de puentes de hidrógeno entre los átomos que forman el enlace peptídico (específicamente entre los grupos carbonilo (-CO-) y amino (-NH-) de los carbonos involucrados en los enlaces peptídicos de aminoácidos cercanos en la cadena). Dependiendo de su estructura secundaria, las partes de la proteína se pueden clasificar en tres clases:

- Hélice alfa: En esta estructura la cadena polipeptídica se desarrolla en forma de helicoide dextrógiro sobre sí misma debido a los giros producidos en torno al carbono beta de cada aminoácido [24].

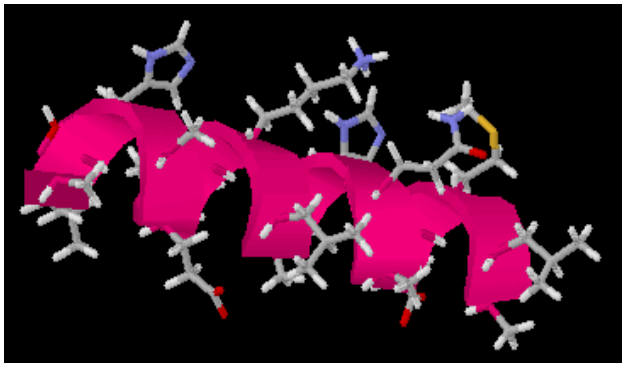


Ilustración 3 Estructura de hélice alfa

- Hoja plegada beta: Cuando el esqueleto polipeptídico se encuentra extendido se adopta una configuración espacial denominada cadena beta. Algunas regiones de proteínas adoptan una estructura en zigzag y se asocian entre sí estableciendo uniones mediante enlaces de hidrógeno intercatenarios. La forma en beta es una conformación simple formada por dos o más cadenas polipeptídicas paralelas (cuyo sentido es el mismo) o antiparalelas (cuyo sentido es opuesto) y se adosan estrechamente por medio de puentes de hidrógeno y diversos arreglos entre los radicales libres de los aminoácidos. Esta conformación tiene una estructura laminar y plegada [24].

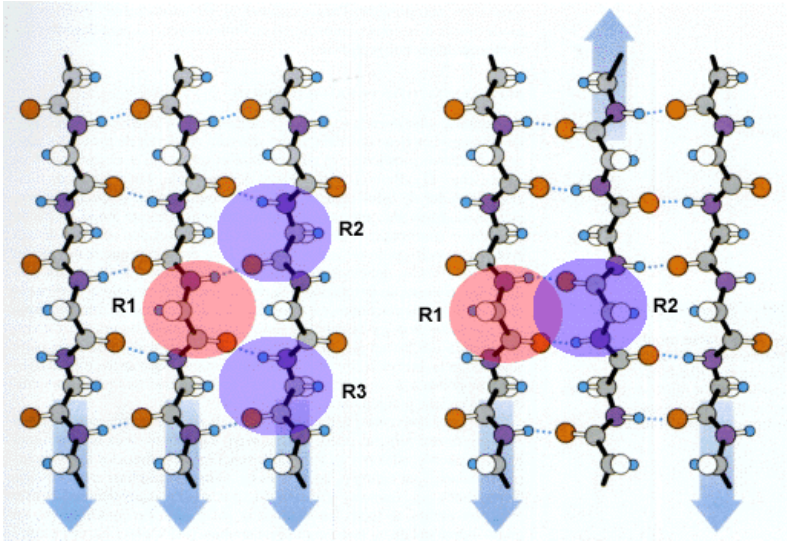


Ilustración 4 Estructura hoja plegada beta

- Giros beta: Conectan secuencias de la cadena polipeptídica con estructura alfa o beta. Son secuencias cortas, con una conformación característica que impone un brusco giro de 180 grados a la cadena principal de un polipéptido [24].

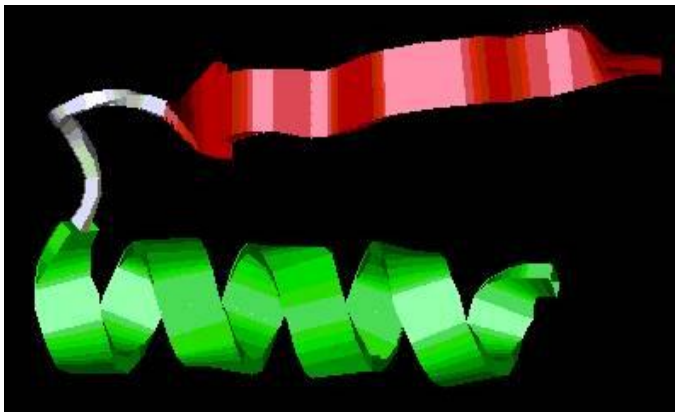


Ilustración 5 Estructura de giros beta

2.5.1 Algoritmos de predicción de la estructura secundaria

Para comparar la estructura secundaria se han desarrollado 3 métodos principales:

2.5.1.1 Algoritmo de Chou-Fasman y derivados

Algoritmo de Chou-Fasman

Estos algoritmos están basados en medidas estadísticas [25] [26]. La idea general es que cada residuo tiene una propensión (P_α^i) a formar una hélice alfa y otra cierta propensión (P_β^i) distinta a formar una hoja beta. Si $P_\alpha^i > 100$, el residuo forma una hélice alfa, si $P_\alpha^i \approx 100$ el residuo es neutral y si $P_\alpha^i < 100$ el residuo rompe una hélice alfa. Las definiciones son similares para P_β^i [27].

1. Predicción de hélice alfa
    a. Determinamos un núcleo de la hélice escaneando grupos de 6 residuos y creando una hélice si encontramos entre los 6 al menos 4 que forman una hélice alfa($P_\alpha^i > 100$) y no más de un residuo que rompa la hélice ($P_\alpha^i < 100$)
    b. Empezamos a alargar el núcleo añadiendo residuos en ambas direcciones hasta que encontremos series con cuatro residuos con media $E[P_\alpha] < 100$
    c. Aceptamos la hélice si su propensión media es $P_\alpha^i > 103$ y $E[P_\alpha] > E[P_\beta]$
2. Predicción de hoja beta
    a. Determinamos un núcleo de la hoja escaneando grupos de 5 residuos y creando una hélice si encontramos entre los 5 al menos 3 que forman una hélice alfa($P_\alpha^i > 100$) y no más de un residuo que rompa la hélice ($P_\alpha^i < 100$)
    b. Empezamos a alargar el núcleo añadiendo residuos en ambas direcciones hasta que encontremos series con cuatro residuos con media $E[P_\alpha] < 100$
    c. Aceptamos la hélice si su propensión media es $P_\alpha^i > 105$ y $E[P_\beta] > E[P_\alpha]$

3. Resolución de conflictos
    a. Si $E[P_\alpha] > E[P_\beta]$ la región es una alfa hélix
    b. Si $E[P_\alpha] < E[P_\beta]$ la región es una hoja beta.

Estos algoritmos traen el problema de que hay un conjunto de coeficientes asignados arbitrariamente puesto que funciona.

Algoritmo probabilístico

Se trata de una mejora del algoritmo de Chou-Fasman que tiene en cuenta además de la estructura, la configuración teniendo en cuenta los ángulos creados por los distintos grupos de residuos. Considera la lista de residuos $[r_1 \ldots r_n]$ y la correspondiente lisa de elementos estructurales $[\zeta_1, \ldots, \zeta_n]$. El elemento ζ_i depende en general de la configuración de un gran número de residuos centrados cerca de la posición i. Hacemos la hipótesis más local y simple posible: el elemento ζ_i depende del triplete de residuos r_{i-1}, r_i, r_{i-1} [28].

Para estimar la estructura computamos la probabilidad condicionada $P(\zeta_i \mid r_{i-1}, r_i, r_{i-1})$, siendo ζ_i la estructura secundaria a predecir y las r los distintos aminoácidos.

Además el elemento estructural en la posición i depende no sólo de los residuos que se encuentran, sino que también depende de los distintos elementos estructurales que están a su alrededor.

Teniendo en cuenta todo esto construimos la hipótesis estructural:

$$\zeta_i^n = \arg \max_{z \in \{H,P,C\}} \{\mathbb{P}(z|r_{i-1}, r_i, r_{i+1})\mathbb{P}(z|\zeta_{i-1}^{n-1}, \zeta_{i+1}^{n-1})\}$$

Donde $i=1,\ldots, N-2$ *y* N es el número de residuos en la proteína.

Esta hipótesis se ejecuta desde la primera estimación (1). Los extremos no pueden ser predichos por este método puesto que no están todos los valores necesarios definidos, por lo tanto hay que buscar otros métodos como sólo usar probabilidades condicionadas a los valores definidos.

2.5.1.2 Algoritmos basados en cadenas de Markov

La versión más básica de este método consiste en considerar la cadena de Markov como un grafo con 3 estados que representan las α-hélices, β-hojas y giros. En este modelo los giros significan cualquier cosa que no sea hélice u hoja. Además tiene un estado de inicio y un estado de fin. La estructura se analiza aminoácido a aminoácido saltando de estado a estado. Para realizar estos saltos se usan probabilidades condicionales, para llegar a un estado depende del estado anterior y del aminoácido de entrada. Por lo tanto se usa P (α|i, β) siendo α el estado objetivo, i el residuo de entrada y β el estado actual. Por supuesto se deben de calcular todas las probabilidades antes de realizar la previsión de la estructura secundaria [28].

Este modelo se puede expandir y se pueden usar varios residuos de entrada, habiendo entonces que calcular las probabilidades. Por ejemplo para parejas de residuos P (α|i, j, β) siendo i, j los residuos.

2.5.1.3 Alineamiento de secuencias de proteínas

Principalmente consiste en usar un algoritmo de alineamiento (como Smith-Waterman) para comparar dos cadenas de proteínas, una con una estructura conocida y otra con una estructura desconocida. Si estas cadenas se parecen entonces es que las dos cadenas de aminoácidos tienen una estructura parecida [8].

2.6 Las bases de datos

Las bases de datos son uno de los motores más notables de la revolución informática. Su existencia ha permitido no sólo el almacenamiento y la gestión de grandes cantidades de datos si no –cosa quizás más importante- su gran accesibilidad debido a la presencia de un modelo de datos y (más o menos) igual para todos.

En la mayoría de las bases de datos ente modelo abstracto es el modelo relacional, introducido por Cobb en 1970, aun así durante los últimos años, han aparecido nuevos modelos que compiten con las bases de datos relacionales y a pesar de que no se espera que sustituyan al modelo relacional, sí se espera que comiencen a usarse cada vez más como alternativa para problemas específicos al modelo relacional.

2.6.1 Comparativa de tipos de bases de datos

Cuando apareció el concepto de bases de datos el único modelo abstracto era el modelo relacional pero recientemente han surgido distintos sistemas de gestión de bases de datos no relacionales, que proponen soluciones diversas a los distintos problemas que pueden llegar a surgir dentro del contexto de la gestión de datos. El hecho de que estos sistemas hayan aparecido no significa que se haya eliminado o vaya a eliminar el modelo anterior, de hecho, todos los distintos modelos siguen conviviendo, y dependiendo del tipo de problema al cual nos enfrentemos, será más óptimo el uso de un modelo u otro [29], por ello en los siguientes puntos podremos ver una comparativa de las bases de datos.

Bases de datos relacionales

Las bases de datos relacionales son sistemas basados en teoría de conjuntos que guardan sus datos sobre tablas bidimensionales implementadas como filas y columnas. Este tipo de bases de datos usan SQL (Structured Query Language) para interaccionar con la base de datos. Los datos que se guardan son explícitamente tipados. Puesto que están basadas en teoría de conjuntos, todas las operaciones que se puedan realizar en conjuntos están definidas en estas bases de datos [29] [30] [31].

Entre las bases de datos relacionales podemos destacar PostgreSQL, MySQL y SQLite3.

Cuando usarlas

Este tipo de bases de datos se utilizan en soluciones donde todos los elementos de un tipo tienen las mismas propiedades y los datos claramente estructurados) [29].

Bases de datos clave/valor

Este tipo de base de datos guarda los datos es pares clave/valor (como su nombre indica), de manera muy parecida a como lo hacen los mapas o diccionarios en varios lenguajes de programación [29] [32].

Las propiedades específicas de la base de datos dependen de la implementación por lo tanto no se puede entrar en tanto detalle como en el caso de las bases de datos relacionales.

Algunas bases de datos destacadas de clave/valor son Redis y Riak [29].

Cuando usarlas

Funciona muy bien en proyectos en los que hay pequeñas lecturas y escrituras frecuentes en modelos de datos simples [32].

Este tipo de base de datos no es bueno en los casos en los que las peticiones a la base de datos son complejas y hay que relacionar los datos. Por lo tanto se suelen usar con grandes cantidades de datos que no requieren peticiones complejas [32].

Bases de datos basadas en columnas

Las bases de datos basadas en columnas guarda sus datos en forma de tabla, pero a diferencia de las bases de datos relacionales (que son orientadas a filas), estas bases de datos son orientadas a columnas, por lo que guardan todos los datos de la columna juntos. Además el añadir columnas es bastante poco costoso [32] [29].

Este tipo de diseño permite que aquellas filas que tenían un dato a nulo (ocupando espacio) en otras bases de datos, ahora no ocupen espacio [29].

Dentro de las bases de datos basadas en columnas destacan HBase, Cassandra, Hypertable [29].

Cuando usarlas

Este tipo de bases de datos está diseñado para trabajar con grandes cantidades de datos en clústeres con múltiples servidores. Por lo tanto se debería usar en aplicaciones con grandes números de datos [32].

El que su principal escalado sea horizontal también permite que sea un tipo de base de datos óptimo para centros de datos distribuidos [33] [34].

Igualmente, por esta distribución de los datos, las aplicaciones deberían poder aceptar una pequeña inconsistencia a corto plazo [34].

Bases de datos basadas en documentos

Este tipo de base de datos tiene la forma de un documento en el cual se indexan datos de distintos tipos, estructuras, u otros documentos [32].

Cada una de estas bases de datos usan una forma distinta de guardar los datos (Comúnmente suele ser XML o JSON(o BSON (Binary JSON))).

Hay muchos tipos de bases de datos basadas en documentos con respecto al diseño, por lo tanto en el momento de elegir una de estas bases de datos hay que estudiar bien los pros y contras entre ellas [29] [34] [33].

Cada una de estas bases de datos tiene una forma de acceso distinta a los datos: por ejemplo MongoDB accede a los datos usando llamadas desde el código del servidor y en algunas de las llamadas usa estructuras JSON, mientras que gran parte de las bases de datos XML usan un lenguaje de dominio específico llamado XQuery que se combina con XPath para formar las consultas.

Algunos ejemplos son MongoDb, CouchDb [29], BaseX y eXist [35].

Cuando usarlas

Este tipo de base de datos es óptima para guardar información compleja no relacionada entre sí y muy variable en términos de estructura. Además realiza búsquedas muy rápidas en complejas estructuras anidadas [32].

Provee de una mayor flexibilidad que las bases de datos relacionales aunque realiza las búsquedas mucho peor [32] [34] [33].

Bases de datos basadas en grafos

Las bases de datos basadas en grafos consisten en un conjunto de nodos y relaciones entre ellos que poseen atributos (clave/valor) en los que se guardan los datos [29], esta manera de guardar los datos permite después realizar operaciones entre datos no jerarquizados pero relacionados mucho más eficientemente en comparación con otros tipos de bases de datos. Este tipo de relaciones permite modelar muchos tipos de escenarios de una manera eficiente y además realizar búsquedas rápidamente a través del grafo.

Las bases de datos basadas en grafos soportan operaciones CRUD (Create, Read, Update and Delete) sobre un modelo de datos de grafos. Además, en general, están diseñadas para aceptar transacciones (OLTP) [36].

En este tipo de bases de datos se ha intentado hacer más eficiente el uso de las relaciones, por lo tanto la aplicación trabajará muy rápido con datos fuertemente relacionados y el aplicaciones que lo que requieren es optimizar la búsqueda sobre las relaciones [37].

El modelaje de los datos es mucho más simple que en otras bases de datos, puesto que consiste en definir los nodos y sus relaciones. Por lo tanto, este tipo de modelo da una gran flexibilidad, y permite aumentar el tamaño y la funcionalidad de la aplicación sin poner en peligro los datos existentes, facilitando así el crecimiento de la base de datos y del mismo modelo [38].

Al usar este tipo de bases de datos nos encontramos con que dada la flexibilidad que se obtiene, permiten un desarrollo incremental e iterativo, permitiendo que se sigan correctamente métodos de desarrollo ágiles [37].

Además, las bases de datos basadas en grafos pueden ser usadas para minería de datos para encontrar patrones en la información guardada en la base de datos, disciplina que está empezando a ser muy importante hoy en día [39].

Varios ejemplos de bases de datos basadas en grafos son Neo4j y Poliglot [29] [38]

Cuando usarlas

Su principal uso es en casos en los que es necesario tener información muy interconectada y se realizan operaciones rápidas sobre las relaciones entre los datos [29] [32] [34] [33].

Además son muy buenas en aplicaciones que tienen opción de crecer, por la flexibilidad que tienen para cambiar y por el desempeño estable que tienen al escalar los datos en comparación con otras bases de datos [37] [40].

Puesto que aceptan transacciones ACID, las cuales son necesarias en gran parte de las aplicaciones actuales, son muy recomendadas para aquellas aplicaciones que necesitan dichas

transacciones además de las características de flexibilidad y desempeño de las que hemos hablado [37].

2.7 Neo4j

2.7.1 La base de datos

Neo4j es un sistema de gestión de bases de datos basadas en grafos [41] [42] que se puede instalar en todas las plataformas (Windows, OS X y Linux) puesto que está implementado en Java. Tiene dos versiones, abierta y comercial (que ofrece más características que la abierta, principalmente en lo referente a la eficiencia, aunque sólo es recomendable para grandes conjuntos de datos).

Para comunicarse con la base de datos se usa una API REST vía HTTP, aunque los desarrolladores también proveen y soportan los drivers necesarios para llamar a la base de datos desde aplicaciones Java, Ruby, Python, PHP, NodeJS y .NET entre otros.

Neo4j para realizar todas las operaciones usa un lenguaje de dominio específico basado en SQL llamado Cypher.

2.7.1.1 Cypher

Cypher es un lenguaje de propósito específico, declarativo y textual usado para realizar peticiones a la base de datos Neo4j [41] [42].Cypher utiliza arte ASCII para indicar las distintas peticiones a la base de datos.

El primer elemento importante de este lenguaje son los nodos que se representan con *()*. Pueden tener un nombre de variable y una etiqueta que se representan así *(nombre_de_variable:Etiqueta)*.

Para unir los nodos entre sí y crear aristas se utilizan relaciones que se representan como una flecha entre dos nodos *()* --> *()*. Además también se pueden añadir etiquetas y nombres de variable como en el caso de los nodos colocando [] entre los guiones. Por lo tanto una representación completa sería *() – [nombre_de_variable:Etiqueta] -> ()*.

Además tanto los nodos como las relaciones pueden tener propiedades que se representan después de la etiqueta entre corchetes con la forma *nombre:valor* , separado por comas .Si unimos todos los componentes tendremos algo así (*nombre1:Etiqueta{prop1: "a*"})-*[r:EtiquetaRel{prop1: "b", prop2:2}]-> (nombre2:Etiqueta{prop1: "b"})*.

Las propiedades pueden usar varios tipos de datos entre los que destacan entero, string y lista de los anteriores [43].

Cypher también tiene varias palabras reservadas entre las que destacan:

- CREATE Crea un nodo con las etiquetas y propiedades especificadas. Un ejemplo de uso es `CREATE (you:Person {name:"You"})`

- MATCH Busca un nodo con las características especificadas, un ejemplo de uso es:

```
MATCH (you:Person {name:"You"})
RETURN you
```

Sobre MATCH hay que destacar que siempre debe de finalizar en RETURN u otra palabra de finalización a diferencia de CREATE

- RETURN Devuelve los argumentos especificados después de él. Un ejemplo de uso es:

```
MATCH  (you:Person {name:"You"})
RETURN you
```

- WHERE Sirve para filtrar los resultados de un MATCH, usa las operaciones >, <, <> =, =~ para comparar los distintos elementos. Un ejemplo de uso es:

```
MATCH (n:Person)-[:KNOWS]->(m:Person)
WHERE n.name = 'Alice'
RETURN n
```

- WITH Permite separar las partes de la petición específicamente, permitiendo declarar las variables que se pasarán a la siguiente parte. Un ejemplo de uso es:

```
MATCH (n:Person)-[:KNOWS]->(m:Person)
WHERE n.name = 'Alice'
WITH m
MATCH (m)-[:LOVES]->(p:Dog)
RETURN p
```

- LIMIT Limita el número de resultados (se suele usar junto a WITH o RETURN). Un ejemplo de uso es:

```
MATCH (n:Person)-[:KNOWS]->(m:Person)
WHERE n.name = 'Alice'
RETURN n
LIMIT 5
```

- CALL Llama a un procedimiento Neo4j con los parámetros que se determinen. Un ejemplo de uso es:

```
CREATE(p1:DNASequence{nucleotides:"ATGGGCAAGGAAGGCCAAATTTAA"})
WITH p1
  MATCH (p2:DNASequence) WHERE p1<>p2
  CALL bio.swexec(p1, p2) return p1;
```

2.7.2 Procedimientos en Neo4j

Un procedimiento es un mecanismo que permite a Neo4j ser extendido mediante código personalizado que puede ser llamado desde Cypher. Todos los procedimientos pueden tener argumentos de entrada, realizar operaciones en la base de datos y devolver resultados.

Los procedimientos se escriben en Java, se compilan en archivos *jar* y se despliegan en la base de datos colocando los *jar* en el directorio *$NEO4J_HOME/plugins* de todos los servidores donde esté la base de datos. Para poder utilizar los procedimientos hay que reiniciar la base de datos [44].

Para llamar a un procedimiento tan sólo hay que ejecutar CALL nombre.procedimiento (argumentos) dentro de una query Cypher que puede ser combinada con otras cláusulas Cypher.

Para crear un procedimiento Cypher se recomienda usar [45] Maven para la creación y gestión del proyecto (aunque no es necesario), aun así el proyecto Maven permite obtener las referencias de una manera fácil y segura, y además se asegura de que descargar la última versión compatible disponible (a no ser que se especifique lo contrario en los ficheros de configuración.

Notas sobre el desarrollo:

- Todos los procedimientos se anotan como `@Procedure`. Si escriben en la base de datos tienen que usar la flag `mode = WRITE`.
- El contexo de un procedimiento, que es el mismo para cada recurso que el procedimiento quiere usar, está anotado en `@Context`.
- Para la input se usa la etiqueta @Name con la cual se nombra un argumento de entrada.

Para testear un procedimiento hay que:

1. Comenzar una instancia de la base de datos.
2. Para cada uno de los test:
    a. Se inicializa el driver.
    b. Se inicializa una sesión
    c. Se realizan los query de test con *sesion.run*
    d. Se comprueba el resultado mediante *ASSERTS*.

3 Análisis

En este apartado hemos explicado las distintas razones de uso de las tecnologías y algoritmos teniendo en cuenta la naturaleza del problema presentado. Específicamente el por qué usamos una base de datos basada en grafos para solventar este problema, específicamente por qué usamos Neo4j, y los algoritmos de Smith-Waterman para el alineamiento de las cadenas y el algoritmo probabilístico para la predicción de la estructura secundaria de las proteínas.

3.1 Por qué usar una base de datos basada en grafos

El problema que se ha querido solventar consiste en tratar datos y obtener relaciones entre ellos. He querido centrarme más en las relaciones que en el contenido de los datos.

Al leer el apartado de cuándo usar las bases de datos relacionales puedes encontrar que están más centradas en los datos y no en las relaciones, por lo tanto, lo que hemos hecho con una base de datos basada en grafos, especialmente las búsquedas a través de las relaciones, habría sido mucho más complicado e ineficiente.

En las bases de datos clave/valor, como ya he especificado, se buscan las pequeñas lecturas y escrituras de datos poco relacionados, por lo tanto en un problema en el que nos centramos en relaciones, no es muy recomendable usarlas.

Hemos descartado las bases de datos basadas en columnas puesto que están más orientadas al movimiento masivo de datos en clústeres, lo cual puede dar a una inconsistencia. Esta inconsistencia no debería estar en una aplicación que tiene que trabajar con todos los datos que se introducen inmediatamente después de ser introducidos.

Sobre las bases de datos basadas en documentos se destaca que son muy rápidas buscando en datos jerarquizados pero no tanto en relaciones entre iguales. Todos los alineamientos, operación que se realiza mucho en nuestra aplicación, es una relación entre iguales, por lo tanto es ineficiente.

Finalmente las bases de datos basadas en grafos permiten muchas interconexiones entre iguales (lo cual es lo que buscamos) y además gran flexibilidad a la hora de añadir datos, además de eficiencia a la hora de escalar, cosa bastante necesaria para una aplicación que aumenta exponencialmente el número de relaciones que tiene respecto al número de nodos con los que trabaja (cada vez que se añada una cadena se crearán relaciones contra todas las demás cadenas del mismo tipo).

Además, dentro de las bases de datos basadas en grafos, elegiremos Neo4j porque al ser la base de datos de este tipo más usada podemos encontrar una mayor cantidad de información y porque ofrece implementar distintos procedimientos, lo cual es muy útil para incluir la funcionalidad necesaria dentro de la base de datos en vez de trabajar con los datos desde fuera, esto nos permitirá dividir las aplicaciones claramente entre la parte específica y la genérica, y por lo tanto crear el ecosistema mencionado en la introducción con mayor facilidad.

3.2 La implementación del algoritmo de Smith Waterman

Hemos elegido una implementación externa del algoritmo de Smith Waterman que consiste en una implementación en C++ con un wrapper en Java (que es el lenguaje que debemos de usar puesto que los procedimientos de Neo4j están escritos en él) [10]

Esta implementación es una extensión de las implementaciones de Striped SW y SWPS3's SIMD para obtener información detallada del alineamiento sin perder eficiencia.

Esta implementación tiene una API que llama a la función de alineamiento.

A pesar de que sé que puede ser un poco más complicado el tratamiento del wrapper (por problemas con Maven) considero que merece la pena puesto que en este tipo de implementaciones es necesaria la eficiencia.

3.3 Selección del algoritmo de predicción de la estructura secundaria de las proteínas.

Como se puede suponer, a pesar de que la solución del alineamiento sería la más sencilla teniendo en cuenta los recursos que tenemos, puesto que vamos a usar el algoritmo de Smith-Waterman para alinear tanto el ADN como las proteínas, el método del alineamiento requiere un gran conjunto de proteínas de las que ya se conoce la estructura secundaria. Recurso que no tenemos, por lo tanto sería demasiado impreciso usar este método.

Respecto a los métodos de Cadenas de Markov y Chou-Fasman, se puede ver que de estos métodos el mejor es Cadenas de Markov puesto que, como explicamos en Chou Fasman, este algoritmo no tiene en cuenta las relaciones entre los distintos residuos, y las Cadenas de Markov, con su expansión sí lo tienen. El algoritmo probabilístico soluciona este problema, puesto que sí que tiene en cuenta las relaciones entre los distintos residuos y además tiene en cuenta el tipo de estructura secundaria que rodea al elemento que estamos prediciendo, por lo que es más exacto. Esta exactitud hace que decidamos usar el algoritmo probabilístico.

4 Extensión de la base de datos Neo4J

De acuerdo con los objetivos del trabajo hemos implementado cadenas de ADN y proteínas como tipos de datos en la base de datos Neo4j y hemos usado los tres algoritmos seleccionados (Smith Waterman, el algoritmo de síntesis de proteínas y el algoritmo probabilístico) para definir operaciones sobre estos datos, que hemos integrado en la base de datos en forma de procedimiento.

Además hemos implementado una aplicación de prueba que realiza distintas operaciones básicas sobre la base de datos Neo4j con el plugin que contiene las extensiones instaladas.

4.1 Instalación de la extensión en una base de datos Neo4j

La instalación de la aplicación es sencilla. Sólo de deben descomprimir los archivos de la extensión en la carpeta plugins de tu instalación Neo4j.

También se debe lanzar el script que proveemos para configurar las restricciones en la base de datos. Dichas restricciones serán explicadas en el apartado 5.

4.2 Uso de la extensión

Para usar las extensiones sólo hay que llamar desde un query Cypher a cada uno de los tres procedimientos usando la instrucción CALL:

bio.swexec: Ejecuta el algoritmo de Smith Waterman para dos cadenas guardadas en nodos de tipo *DNASequence*, *AminoacydSequence* o *Structure* en los atributos *nucleotides*, *aminoacyds* o *structure* respectivamente. La entrada del método son dos nodos de estos tipos (es indiferente el orden) y no tiene salida.

bio.proteinTranslation: Ejecuta la traducción de ADN a proteínas de un nodo de tipo DNASequence con un atributo llamado *nucleotides*. La entrada es dicho nodo y no tiene salida

bio.proteinStructure: Ejecuta la predicción de la estructura secundaria de las proteínas. La entrada es un nodo de tipo *AminoacydSequence* con un atributo *aminoacyds* y no tiene salida.

4.3 Puesta en marcha de la aplicación de prueba

Para lanzar la aplicación de prueba lo que se debe hacer es:

1. Copiamos los archivos de la aplicación al directorio deseado.
2. Editamos el archivo de configuración config.js en server. Aquí podemos configurar el host, el usuario (*config.database.user*) y la contraseña (*config.database.pass*) de la base de datos. También se puede configurar el puerto en el que se despliega el servidor (*config.database.port*).
3. Entramos a la carpeta *app_prueba* y lanzamos *npm start*.

La interfaz de la aplicación está dividida en 4 partes principales:

Arriba a la izquierda se ve un panel blanco que es un canvas en el que se dibujarán los grafos.

Arriba a la derecha tenemos un menú con las opciones de operaciones que podemos elegir.

Abajo a la izquierda tenemos un espacio en blanco que se rellena con información al pasar por encima del grafo y con el tipo y un mensaje sobre el estado de la última petición realizada.

Abajo a la derecha tenemos un espacio en blanco que se rellena con formularios al elegir una operación del menú que hay encima.

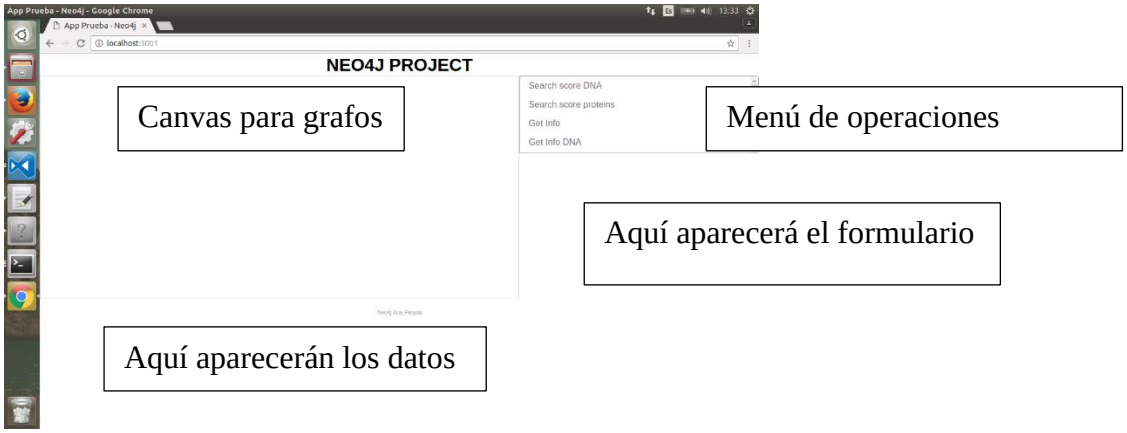


Ilustración 6 Ejemplo de interfaz de aplicación

Los grafos aparecen después de lanzar una de las operaciones, como manera de representar los datos. Los posibles grafos son:

- Grafo de alineamiento: El grafo de alineamiento se genera cuando se piden nodos de un mismo tipo, y por lo tanto tienen todos los nodos el mismo valor. Todos los nodos son grises, tienen posiciones aleatorias y un score asignado a las aristas.
- Grafo de información: Este grafo aparecerá cuando se quieren representar relaciones entre nodos de distintos tipos (se suele representar el nodo de información, los de ADN relacionados, los de proteínas asociadas y la estructura). Se representan con nodos de distintos colores, morado para los nodos de información, azul para los nodos de ADN, naranja para los nodos de proteínas y negro para los nodos de secuencias.

El pulsar dos veces sobre un nodo permite rellenar todos los campos en los que se puede introducir ese tipo de dato. En el caso de un grafo de alineamiento rellena tanto lo de cadenas del tipo de dato (ADN y proteínas) y las ids de la información.

Las posibles operaciones son:

- ***Search Score DNA***: Permite buscar todas las relaciones con un score mínimo entre los nodos que siguen un patrón de ADN. La salida en un grafo de tipo alineamiento.
- ***Search Score proteins***: Muy parecido al anterior, pero busca en los nodos de proteínas.
- ***Get Info***: Devuelve información en forma de un grafo de información, con toda la información relacionada con un nodo de información que se busca por id.
- ***Get Info DNA***: Parecido al anterior pero busca sobre un patrón de nucleótidos de ADN.
- ***Get Info Protein***: Parecido al anterior pero busca sobre un patrón de aminoácidos.
- ***Get All Info***: Devuelve todos los nodos de información como un nodo de información.
- ***Add DNA Sequence***: Añade una secuencia a la base de datos con un nodo de información (en forma de id) asociado.
- ***Add Info***: Añade un nodo de información con un nombre, un nombre científico y los codones de inicio.

- ***Populate***: Añade varios nodos desde un fichero y los asocia con un nodo de información introducidos en forma de id.

Para más detalle sobre el uso de estas operaciones ver Anexo II

5 Diseño e implementación

En este apartado se indicará cómo se ha desarrollado el sistema así como una aplicación de prueba.

5.1 Estructura de la base de datos

Para realizar esta aplicación hemos generado una estructura de nodos y relaciones que será con la que trabajarán tanto los procedimientos como la aplicación.

Hemos de señalar que la estructura que proponemos aquí no es la única con la que puede ser lanzada la aplicación, si no la más básica, todas las extensiones funcionarán siempre y cuando se mantenga esta estructura en la base de datos y se le añadan más cosas si se quiere.

Tipos de nodos

1 DNASequence: Guarda una cadena que contiene ADN

Propiedades:

nucleotides: guarda la cadena de ADN en dirección 5' - 3'

2 AminoacydSequence: Guarda una cadena perteneciente a un aminoácido

Propiedades:

sequence: Secuencia de proteínas

3 Structure: Guarda una cadena que contiene la estructura secundaria de la proteína

Propiedades:

structure: estructura de la proteína

4 Info: Guarda distintos tipos de información para las cadenas como propiedades.

Propiedades:

startCodons: lista que indica los codones de inicio para la transformación de ADN a proteínas.

(opcional) *name*: nombre del organismo al que pertenece la muestra

(opcional) *sciname*: nombre científico del organismo al que pertenece la muestra

Tipos de relaciones

1 ALLIGNS

Asociación entre nodos del mismo tipo (excepto *Info*) que compara las secuencias (de ADN o proteínas)

Atributo:

score: score del algoritmo de Smith Waterman

2 TRANSLATES

Asociación entre un nodo de tipo *DNASequence* y su respectivo de tipo *AminoacydSequence*

3 STRUCTURES

Asociación entre un nodo de tipo *AminoacydSequence* con su respectivo de tipo *Structure*

4 INFO

Asociación de un nodo de ADN con un nodo de información

Restricciones

Para hacer que la base de datos tuviera sentido y consistencia hemos decidido añadirle algunas restricciones que están contenidas en un script:

Restricción sobre la cadena de nucleótidos, no puede haber dos cadenas iguales.

Restricción sobre la cadena de aminoácidos, al igual que en el caso de cadenas de ADN no puede haber dos iguales, si dos cadenas de ADN que generan la misma cadena de aminoácidos habrá 2 relaciones, entre el nodo de aminoácidos y cada una de las cadenas de ADN, aunque sólo haya un nodo con el objetivo de ahorrar tanto espacio (para evitar nodos duplicados) como tiempo, al evitar alineamientos innecesarios.

Restricción sobre la estructura. Al igual que en el caso de la cadena evitamos que se creen dos iguales, aunque después permitimos que se creen relaciones con varias cadenas de aminoácidos con el objetivo de ahorrar tiempo y espacio.

4.2 Proyecto de procedimiento Neo4j

El proyecto será realizado en Java usando el software de gestión de proyectos Maven. Este software permite dividir el proyecto principal en

El proyecto está constituido como un proyecto Maven con distintos módulos y uno que los une todos.

Dentro del proyecto podemos encontrar 4 carpetas:

- **plugins**: No pertenece al proyecto Maven aunque es necesaria. Contiene todos los recursos externos no java dentro de la carpeta *external*. Estos recursos son las implementaciones de los algoritmos de Smith-Waterman(C/C++) y de predicción de la estructura secundaria (Python) mencionados anteriormente.

- **protein_structure**: Contiene un wrapper del recurso Python que contiene el algoritmo de predicción de la estructura secundaria.
  Usa el Jython [46] para hacer la unión con Java, para usar el código Python se ha tenido que proporcionar una dirección relativa respecto a la carpeta de ejecución en la que se guardan los archivos Python y que tiene que estar en la misma dirección relativa en todos los sistemas en los que se lance la extensión (tanto en los test como en la base de datos). Por esta misma razón existe la carpeta *plugins*, *plugins* es el nombre de la carpeta

donde se guardan las extensiones en Neo4j, y para mantener la estructura y el orden ya impuesto se decidió usar esta estructura de ficheros.

- **translation:** Módulo Maven propio que realiza la traducción de ADN a proteínas. Contiene principalmente dos archivos:

  - **CodonTable**: contiene una lista de elementos guardados en una tabla (guardada como HashMap) con los codones y las traducciones. Tiene la función *getAminoAcidSequence* que obtiene la proteína respectiva dado un codón.

  - **Translator**: Realiza la traducción de ADN de una cadena con dirección 5'-3', tiene dos funciones:

    - **translate**: su entrada es una cadena de ADN y la lista de codones. Esta función compara todos los codones con los codones de inicio y cuando un codón de inicio se encuentra se introduce una metionina (M) en la cadena como primer carácter y se obtienen las demás proteínas de la tabla de *CodonTable* y se añaden a la cadena de aminoácidos. Cuando la tabla no devuelve nada (null) implica que ha finalizado la traducción y se añade la cadena a una lista y se vuelve a empezar a buscar el codón de inicio. Finalmente se devuelve la lista de cadenas de aminoácidos.

    - **translation**: su entrada es la cadena de ADN y una lista de codones de inicio. Realiza 3 llamadas a la función *translate*, una con la cadena de ADN tal y como entra, otra con la cadena sin contar el primer nucleótido y otra sin contar los dos primeros nucleótidos. Cada una de las cadenas de aminoácidos obtenidas se introducen en un array y se devuelven.

  - **blast-procedure**: En esta carpeta se unen todos los módulos anteriores y se crean los distintos procedimientos. Contiene tres archivos principales:
    - **BlastLabel**: Enumerado de tipos de nodo que se usarán en la base de datos (*DNASequence, AminoacydSequence, Structure* e *Info*)

    - **BlastRelationshipType**: Enumerado de los tipos de relaciones que se usan en la base de datos.(*ALLIGNS*, *INFO, TRANSLATES*, *STRUCTURES*)

    - **Blast**: Contiene 3 funciones principales, cada una asociada a una de las extensiones creadas:

      - **swexec**: La entrada de esta función son 2 nodos a alinear que deben de ser del mismo tipo (*DNASequence*, *AminoacydSequence*, *Structure*) Lo primero que se hace en esta función es obtener la cadena del nodo (*nucleotides, aminoacyds* o *structure*). Se diseñó esto de esta manera puesto que así es mucho más intuitivo para el usuario a la hora de añadir nodos, además esto obliga a que los pares de nodos tengan el mismo tipo, puesto que se comprueba en tipo de ambos a la vez. Posteriormente se hacen dos comprobaciones, se comprueba si los nodos son iguales, caso en el que no se creará una arista hacia sí mismo, y si existe la relación entre los nodos, caso en el que no se creará otra nueva. Después crea las estructuras necesarias para el alineamiento y llama a la función *allign* del apartado de

alineamiento. Finalmente crea las relaciones y añade los scores a las relaciones.

- **proteinTranslation**: A este procedimiento sólo se le introduce como entrada un nodo de tipo *DNASequence*. Lo primero que hace es obtener la cadena de la propiedad *nucleotides*. Posteriormente se obtienen las relaciones de tipo INFO que tiene el nodo y se obtienen los codones de inicio guardados en *Start Codons*. Se llama a la función de traducción (*translation*) del módulo *translation*. Finalmente, para la lista de listas que devuelve esta función se crea para cada una de ellas un nodo con la propiedad *aminoacyds* siendo la cadena obtenida, si no existe. Después se crea una relación entre el nodo de ADN y el de aminoácidos si no existe dicha relación.

- **proteinStructure**: Este procedimiento es muy parecido al de traducción, al igual que en él se introduce un nodo como argumento, en este caso debe de ser de tipo *AminoacydSequence*. Posteriormente se obtiene la secuencia de aminoácidos de *aminoacyds* y se llama a la función *pred* de *ProteinStructure* (del módulo *proteinStructure* ya explicado). Finalmente, al igual que en el otro método se crean el nodo y la relación si no existen.

Además se han realizado distintos tests unitarios para las 3 extensiones y los módulos *protein_structure* y *translation*.

4.3 Aplicación de prueba

Para realizar esta aplicación hemos usado Javascript junto con HTML5 y CSS3 para el frontend usando el framework de CSS PureCSS [47], además para mostrar los grafos hemos usado la biblioteca de Javascript SigmaJS [48] que permite mostrar grafos de una manera sencilla. Para el backend hemos usado NodeJS [49] con el framework de ExpressJS [50], que permite una redirección más sencilla que usando Javascript puro, y el driver que provee el desarrollador de Neo4j, por supuesto como base de datos hemos usado nuestra base de datos Neo4j con los procedimientos creados.

En este apartado vamos a explicar parte de las características técnicas de la aplicación, si se desea conocer cómo se usa la aplicación diríjase al apartado de tutorial.

Todos los apartados de la aplicación se han desarrollado de una manera muy parecida con la excepción de la llamada específica a la base de datos que en general es la que realiza la acción buscada y el apartado de *Populate*.

Para ver un estudio específico de cada uno de los querys que usa la aplicación, en el apartado de Pruebas (5.2) se han probado dichos querys con pequeñas variaciones para evitar usar variables de la aplicación y en este mismo apartado se señala en qué opción de la aplicación se usan.

Para crear el HTML se han utilizado el motor de plantilla de Jade [51] desde ExpressJS, permitiendo que se generaran las distintas partes de la interfaz de manera dinámica.

Como se puede ver en el tutorial, la aplicación tiene partes estáticas como son el menú superior, los formularios (que a pesar de que aparecen dependiendo de la opción seleccionada del menú

superior se descargan sólo una vez) o los distintos contenedores. Además tiene varias partes dinámicas que se actualizan usando AJAX.

Para los mensajes de comunicación entre el servidor y el cliente se han usado varios tipos comunes que se encuentran en un archivo Javascript que es usado tanto en el servidor como en el cliente. Estos tipos son *GraphType*, que puede tomar los valores *ALLIGN, DEPTH* y *NONE* e indica el tipo de grafo que se va a representar (de alineación, de profundidad o ninguno) y *AllignType* que puede tomar los valores *DNA* y *AMINO* e indica el tipo de dato que se va a guardar en un grafo de tipo *ALLIGN*.

La mecánica general de uso de la aplicación consiste en rellenar un formulario dentro de la interfaz, esta mandará los datos de la petición al servidor y el servidor validará y parseará los datos para finalmente hacer la llamada a la base de datos correspondiente y devolver varias filas de datos. Estos datos serán parseados y colocados en distintas estructuras dependiendo de la operación y además se devolverá en tipo de grafo que posteriormente se representará en el campo *type*.

En el caso de "*Get Info DNA*", "*Get Info*", "*Get Info Protein*" y "*Get All Info*" se devolverá una lista con todas las relaciones posibles en el campo *info* y el campo *type* tomará el valor de *GraphType.DEPTH*.

En el caso de "*Search Score DNA*" y "*Search Score Proteins*" se devolverán dos listas, *scores* con todas las aristas que devuelve la base de datos y *nodes* con todos los nodos que son extremos de las aristas de scores. El campo *type* tomará el valor *GraphType.ALLIGN*. Además contará con el campo *allignType* el cual tomarán los valores *AllignType.DNA* para "*Search Score DNA*" y *AllignType.AMINO* para "*Search Score Proteins*".

Finalmente, para todos los casos en los que se añaden datos a la base de datos o todos los mensajes de error el campo *type* tomará el valor *GraphType.NONE.*

Además de esto todas las operaciones devolverán un mensaje dando un poco de información sobre el estado de la operación ejecutada.

Populate permite subir un fichero de texto a la aplicación y además chequea que es de conjuntos de ADN. Para hacer esta característica hemos usado una extensión de ExpressJS llamada Multer [52], que permite subir ficheros y usarlos en el servidor. Después se han extraído las distintas cadenas guardadas en el archivo y se han parseado, para finalmente añadirlas a la aplicación.

La parte de la visualización de los grafos se ha realizado con la biblioteca SigmaJS.

SigmaJS es una biblioteca de JavaScript que permite visualizar en un Canvas distintos tipos de grafos, con sus diversas características.

Hemos configurado la biblioteca de manera que la aplicación muestre un conjunto de nodos y permita arrastrarlos. También es capaz de guardar datos asociados a cada uno de los elementos del grafo y mostrarlos al hacer hover al igual que es capaz de rellenar los distintos campos del formulario como las ids de los nodos o las cadenas de aminoácidos y nucleótidos para luego poder hacer las peticiones.

Representaremos dos tipos distintos de grafos:

Grafo de alineación: es el que aparecerá cuando busquemos los datos por scores (ADN y proteínas), en este caso todos los nodos son iguales y aparecerán en gris y en distintas posiciones aleatorias.

Grafo de profundidad: es el que muestra la asociación de una información. Muestra nodos de información asociados con nodos de ADN, proteínas y estructuras. Cada uno de los distintos tipos de nodo aparecen en un color distinto.

5 Pruebas

5.1 Pruebas sobre la base de datos

En los apartados siguientes hemos realizado varias pruebas sobre la eficiencia de las querys.

Es importante para valorar los tiempos obtenidos que las ejecuciones se han medido en una base de datos Neo4j instalada en un sistema que cuenta con un procesador Intel(R) Core(TM) i5-3230M de 2.60GHz, una RAM de 8GB y aproximadamente 10GB de espacio libre en disco duro. Las pruebas se han realizado en un entorno con un sistema operativo Ubuntu.

Con estas especificaciones del sistema de desarrollo nos encontraremos que si se despliega la aplicación en un entorno de producción los tiempos serán más bajos y la eficiencia mejor, por lo tanto los resultados aquí destacados y explicados son sólo orientativos.

5.1.1 Estudio de la eficiencia de las querys que se usan en la aplicación

Aquí tenemos todas las llamadas Cypher usadas por la aplicación en las cuales hemos sustituido los scores y los patrones de aminoácidos y nucleótidos para realizar la llamada a la base de datos usando datos controlados por nosotros, por lo tanto son peticiones muy parecidas a las que se harán en cualquier aplicación específica que se cree usando esta extensión. Por lo tanto, estos son ejemplos de las posibles peticiones más comunes a la hora de acceder a la base de datos.

Para probar la eficiencia de las querys utilizadas hemos usado la operación PROFILE con estas querys en una base de datos que contiene 400 cadenas de distintas longitudes cercanas a 400 nucleótidos las cuales fueron generadas aleatoriamente mediante un script. Hemos de añadir que se ha intentado añadir más nodos y había un momento con un número alto de nodos que si se hacía una query muy exigente (que trabaja con un número grande de datos) aparecía un error puesto que la memoria virtual de la Java Virtual Machine estaba llena. Esto se debe a la poca capacidad del hardware de pruebas y algunas variables específicas de la Java Virtual Machine, pero es poco probable que ocurra si se despliega el sistema en un servidor adecuado.

A continuación podemos ver el nombre de la tarea que realiza esta query, la query de prueba que hemos ejecutado, el tiempo de ejecución, su esquema de ejecución y una pequeña explicación de dicho esquema.

Dentro de las explicaciones de los apartados siguientes se harán referencias a números entre paréntesis (1), esto son referencias a la imagen que se ve encima de la explicación donde aparecen dichas referencias.

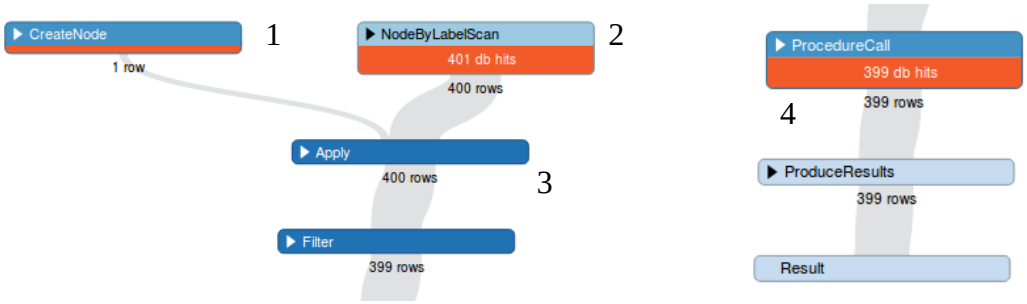
Añade ADN

Usado en la opción de la aplicación *Add DNA Sequence*

```
CREATE(p1:DNASequence{nucleotides:"ATGGGCAAGGAAGGCCAAATTTAA"})
WITH p1
  MATCH (p2:DNASequence) WHERE p1<>p2
  CALL bio.swexec(p1, p2) return p1;
```

Tiempo de ejecución: 571 ms

Esta query crea un nodo de ADN con nucleótidos asignados en nucleotides y posteriormente busca todos los nodos de ADN que no hayan sido creados por la query y hace la llamada a bio.swexec.



Esquema 1 Añade ADN

Como podemos ver en la imagen se crea un nodo en (1), y después se buscan entre todos los nodos los que no han sido creados en este momento, lo cual está representado en (2) por esta razón en (3) hay 400 filas y después de (4) 399. La ejecución de swexec se ha realizado 399 veces, una por cada uno de los nodos que se van a alinear con el creado. Tiene sentido el tiempo que tarda esta ejecución puesto que requiere trabajar con un número bastante alto de datos y además es una operación de inserción, que son mucho más pesadas que las de búsqueda.

Obtiene aristas de ADN alineadas por calificación

Usado en la opción de la aplicación *Search score DNA*

```
MATCH (p1:DNASequence)-[r:ALLIGNS]->(p2:DNASequence)
      WHERE r.score > 100 AND p1.nucleotides =~ ".*"
      return ID(p1) AS NodeId1,  ID(r) AS EdgeId,
      r.score as score, ID(p2) AS NodeId2
```

Tiempo de ejecución: 638 ms

En esta petición Cypher podemos ver que se buscan todas las relaciones de tipo ALLIGN entre dos nodos de tipo ADN, filtra las relaciones entre los nodos para mantener sólo las que tengan un score mayor de 100 y los nucleótidos del primer nodo sigan una expresión regular. Esto es muy útil para obtener las distintas relaciones que llegan a un nodo.

Esquema 2 Obtiene aristas de ADN alineadas por calificación

En este caso podemos ver cómo primero busca el primer nodo (1) y luego obtiene el segundo a través de la relación (2), como el grafo de la base de datos que asocia a los nodos de tipo ADN es un grafo completo el número de arista será n^2 siendo n el número de vértices. Finalmente se filtra por scores y cadena de nucleótidos para obtener la salida final (3). Son

comprensibles los 638 ms que tarda puesto que hace una operación equivalente a un producto cartesiano al buscar sobre todas las relaciones de un grafo completo, por lo tanto es costosa la operación.

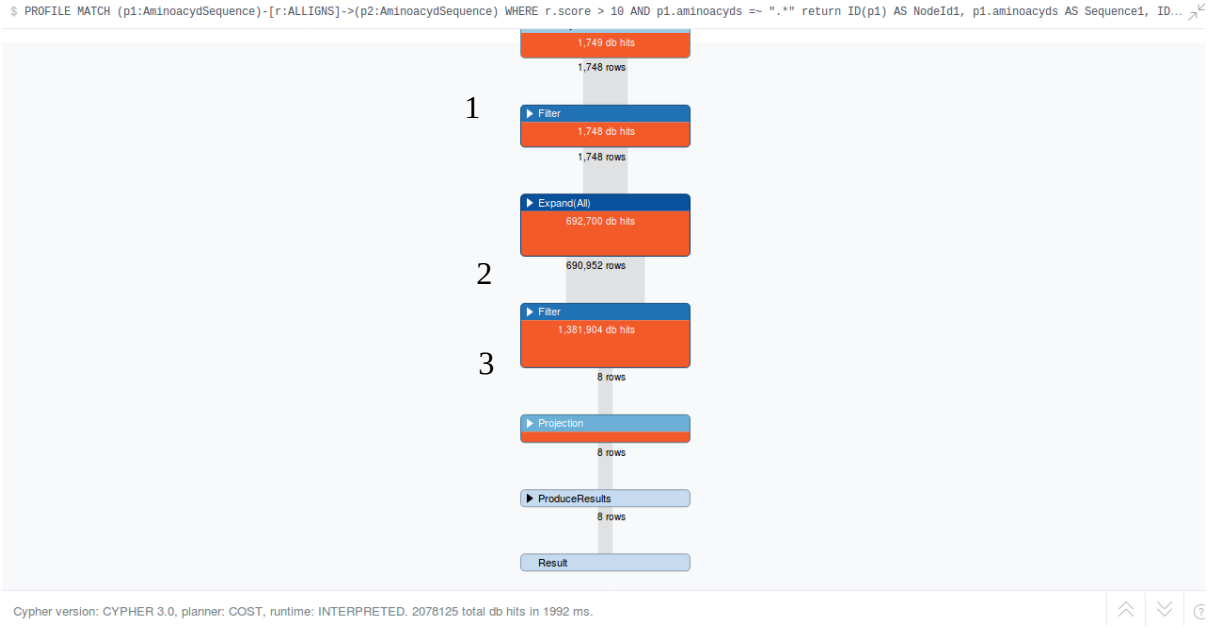
Obtiene aristas de proteínas alineadas por calificación

Usado en la opción de la aplicación Search score proteins

```
MATCH (p1:AminoacydSequence)-[r:ALLIGNS]->(p2:AminoacydSequence)
        WHERE r.score > 10 AND p1.aminoacyds =~ ".*"
        return ID(p1) AS NodeId1,  ID(r) AS EdgeId,
        r.score as score, ID(p2) AS NodeId2
```

Tiempo de ejecución: 1992 ms

Este query es análoga a la anterior, solo que en vez de trabajar con ADN trabaja con proteínas.



Esquema 3 Obtiene aristas de proteínas alineadas por calificación

Al igual que en el caso anterior, primero se busca el primer nodo(1) y luego se expande la cantidad de datos siguiendo las relaciones(2), para finalmente filtrar por score (3). Pero a diferencia del caso anterior que había 400 filas de datos en el paso (1), en este caso hay 1748, puesto que al traducir a aminoácidos se genera más de un resultado. El que haya más resultados en la base de datos implica que el coste sea mayor que en el caso anterior.

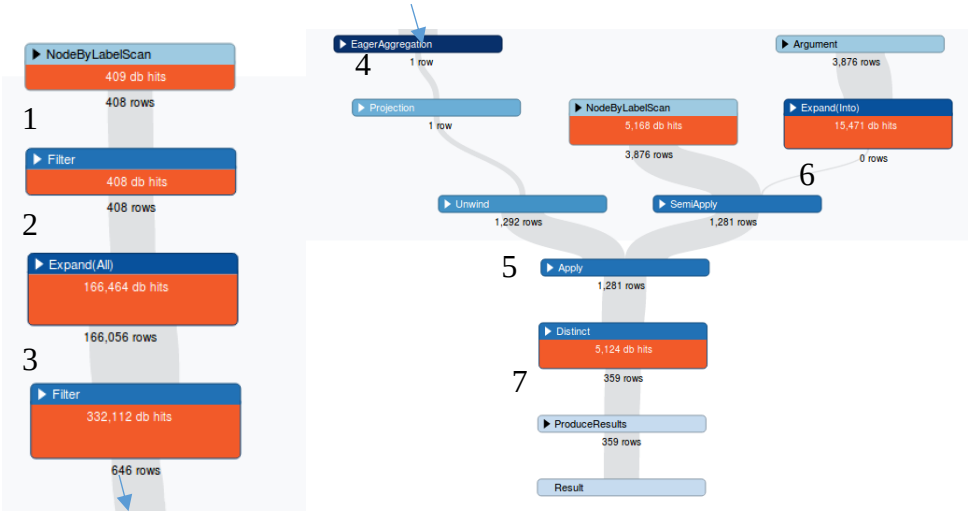
Obtiene nodos de ADN alineado por calificación

Usado en la opción de la aplicación *Search score DNA*

```
MATCH (p1:DNASequence) -[r:ALLIGNS]-> (p2:DNASequence)
        WHERE p1.nucleotides =~ ".*" AND r.score > 100
        WITH collect(p1) + collect(p2) as p
        UNWIND p as dna
        MATCH (i:Info) WHERE (dna)-[:INFO]-> (i)
        RETURN distinct ID(dna) AS NodeId1, dna.nucleotides AS Sequence1,
        i.name AS Name, i.sciname as SciName,
        i.startCodons AS StartCodons, ID(i) AS InfoId
```

Tiempo de ejecución: 15496 ms

Este query encuentra todos los nodos en relaciones de tipo *ALLIGN* que tienen un score mínimo definido y uno de los nodos cumple que sus nucleótidos siguen una expresión regular. Posteriormente se crean dos listas con los nodos (usando *collect*) unen las dos listas para formar otra y esta última lista se separa (usando *UNWIND*) creando la lista *dna* y se busca la información asociada a todos los nodos de la lista *dna* y se devuelve la información del nodo de ADN y del nodo de información.



Esquema 4 Obtiene nodos de ADN alineado por calificación

Se puede ver que en (1) se obtienen todos los nodos que cumplen en filtro de que sus nucleótidos cumplen la expresión regular ".*" y posteriormente se expanden obteniendo todas las asociaciones entre los nodos obtenidos en (1) entre todos los nodos de la base de datos, esto se ve en (2). Obtener estas asociaciones es bastante costoso puesto que al ser un grafo completo implica trabajar con $O(n^2)$ aristas habiendo n nodos. En (3) se filtra todo este conjunto de aristas para obtener sólo aquellas cuyo score sea mayor de 100. Después en (4) se crean dos listas de nodos (una por el nodo de inicio de la relación y una por el nodo de fin) y se unen, para luego desplegarse, lo cual vemos en (5). En (6) es donde se encuentran y unen los nodos de tipo información asociados a los nodos de la lista. Finalmente, en 7, se devuelve información de todos los nodos siempre que no esté repetida, lo cual se ve al reducirse bastante el número de filas devueltas.
Se ve que el tiempo necesario para ejecutar esta petición es bastante alto, esto es porque hay que obtener y filtrar todas las aristas de la base de datos (en este caso) entre otras operaciones con grandes cantidades de datos.

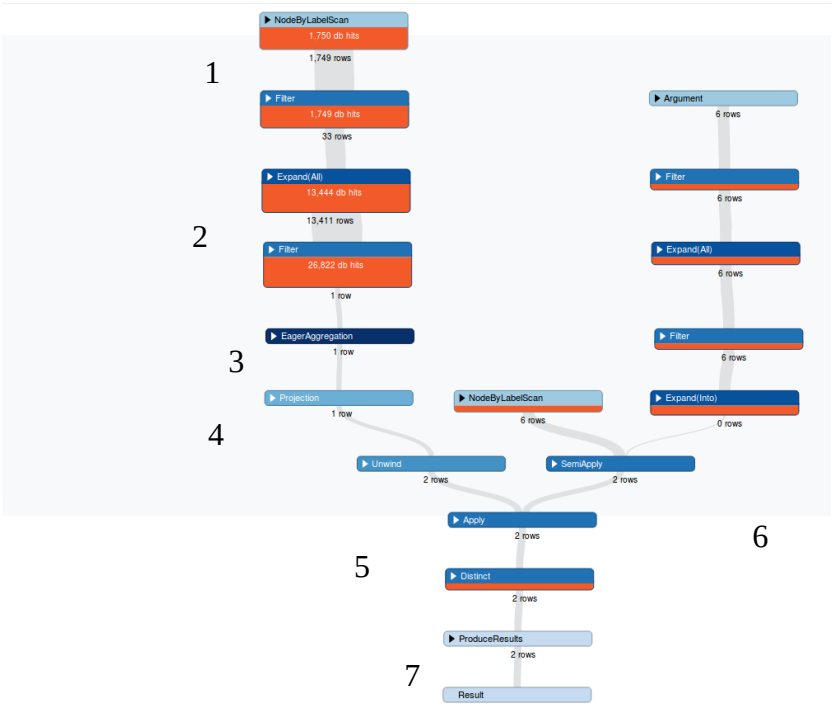
Obtiene nodos de proteínas alineadas por calificación

Usado en la opción de la aplicación *Search score proteins*

```
MATCH (p1:AminoacydSequence) -[r:ALLIGNS]-> (p2:AminoacydSequence)
        WHERE p1.aminoacyds =~ ".*MM.*" AND r.score > 10
        WITH collect(p1) + collect(p2) as p
        UNWIND p as amino
        MATCH (i:Info) WHERE (amino:AminoacydSequence)<-[:TRANSLATES]-
(:DNASequence)-[:INFO]->(i)
         RETURN distinct ID(amino) AS NodeId1, amino.aminoacyds AS Sequence1,
i.name AS Name, i.sciname as SciName,
         i.startCodons AS StartCodons, ID(i) AS InfoId
```

Tiempo de ejecución: 95 ms

Este query encuentra todos los nodos en relaciones de tipo *ALLIGN* que tienen un score mínimo definido y uno de los nodos cumple que sus aminoácidos siguen una expresión regular. Posteriormente se crean dos listas con los nodos (usando *collect*) unen las dos listas para formar otra y esta última lista se separa (usando *UNWIND*) creando la lista amino y se busca la información asociada a todos los nodos de la lista amino y se devuelve la información del nodo de aminoácidos y del nodo de información.



Esquema 5 Obtiene nodos de proteínas alineadas por calificación

Este caso es muy parecido al caso anterior, tan sólo compara por aminoácidos, por lo que complica un poco más la lógica a la hora de acceder a la información. Se puede ver que en (1) se obtienen todos los nodos que cumplen en filtro de que sus aminoácidos cumplen la expresión regular ".*MM.*" y posteriormente se expanden obteniendo todas las asociaciones entre los nodos obtenidos en (1) entre todos los nodos de la base de datos, esto se ve en (2). Como ya hemos explicado antes, obtener estas asociaciones es bastante costoso puesto que al ser un grafo completo implica trabajar con $O(n^2)$ aristas habiendo n nodos. En (3) se filtra todo este conjunto de aristas para obtener sólo aquellas cuyo score sea mayor de 10. Después en (4) se crean dos listas de nodos (una por el nodo de inicio de la relación y una por el nodo de fin) y se unen, para luego desplegarse, lo cual vemos en (5). En (6) es donde se encuentran y unen los nodos de tipo información asociados a los nodos de la lista, se puede ver que para encontrar los nodos en este caso, a diferencia del anterior se busca en dos pasos (primero se busca la relación de *TRANSLATES* entre el nodo de aminoácidos y el de ADN y posteriormente la de INFO entre el nodo de ADN y el de información o viceversa, según el esquema de la query no está claro. Finalmente, en (7), se devuelve información de todos los nodos siempre que no esté repetida, lo cual se ve al reducirse bastante el número de filas devueltas.

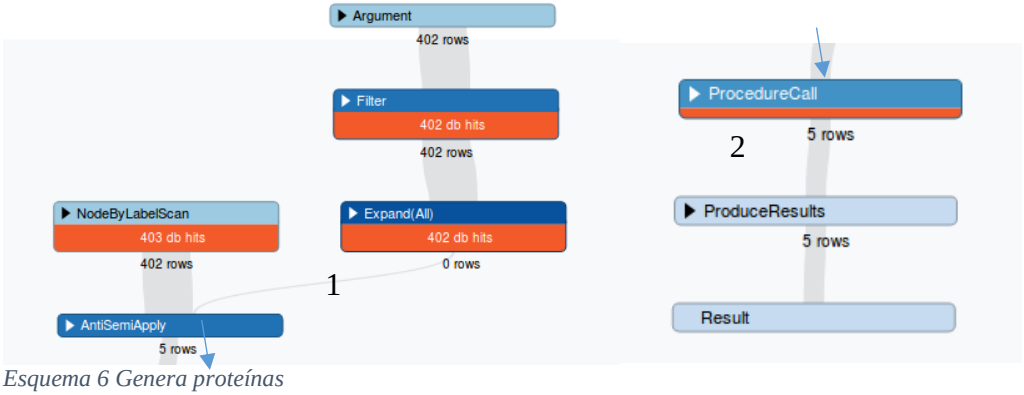
Esta petición, a diferencia de la anterior, tarda un tiempo razonable, esto es por el pequeño filtro que se realiza a los nodos origen de la relación, por lo tanto se calculan muchas menos relaciones, lo que mejora la eficiencia mucho.

Genera proteínas

Usado en la opción de la aplicación *Populate* y *Add DNA Sequence*

```
MATCH (p1:DNASequence) WHERE NOT (p1:DNASequence) -[:TRANSLATES] -> ()
        CALL bio.proteinTranslation(p1) return p1
```

Tiempo de ejecución: 60 ms
Esta ejecución busca todos los nodos de ADN que no tienen una relación de tipo *TRANSLATES* y posteriormente ejecuta el procedimiento de traducción de proteínas.



Esquema 6 Genera proteínas

Como se puede ver, lo primero que se ha hecho en (1) es filtrar la búsqueda de las secuencias de ADN por aquellas que tienen una relación de tipo *TRANSLATES* y posteriormente, en (2), se ejecuta el procedimiento con los elementos ya filtrados.
Sobre la eficiencia de este query hay que decir que en un inicio se buscaron los nodos que se relacionaban con otros, en vez de aquellos que tienen una relación, y esto era mucho más ineficiente puesto que generaba todas las combinaciones en vez de filtrar.
Se puede ver cómo esta ejecución es muy eficiente gracias al filtrado mencionado.

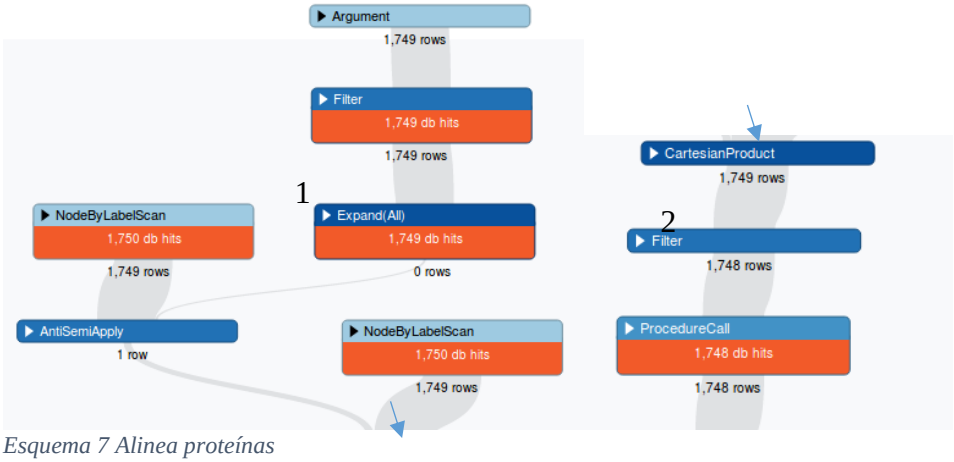
Alinea proteínas

Usado en la opción de la aplicación Populate y Add DNA Sequence

```
MATCH(p1:AminoacydSequence), (p2:AminoacydSequence) WHERE p1<>p2
        AND NOT (p1:AminoacydSequence) -[:ALLIGNS] -> ()
        CALL bio.swexec(p1, p2) return p1
```

Tiempo de ejecución: 599 ms

En este caso se vemos una petición Cypher en la cual selecciona todos los nodos de tipo aminoácido para dos variables distintas, lo cual implica un producto cartesiano. Posteriormente, entre las parejas generadas, se filtran aquellos nodos que son iguales y aquellos en los cuales el primer nodo no tiene relaciones de tipo *ALLIGN*, para finalmente ejecutar *bio.swexec* y proceder al alineamiento de cadenas.



Esquema 7 Alinea proteínas

En el esquema se puede ver cómo lo primero que se aplica es el filtro sobre la relación(1), seleccionando aquellos nodos que no tiene relaciones de tipo *ALLIGN*. Posteriormente, con los resultados filtrados, es cuando se obtiene el producto cartesiano (2), por lo tanto nunca se genera un gran número de filas, por lo que puede llegar a ser muy eficiente.

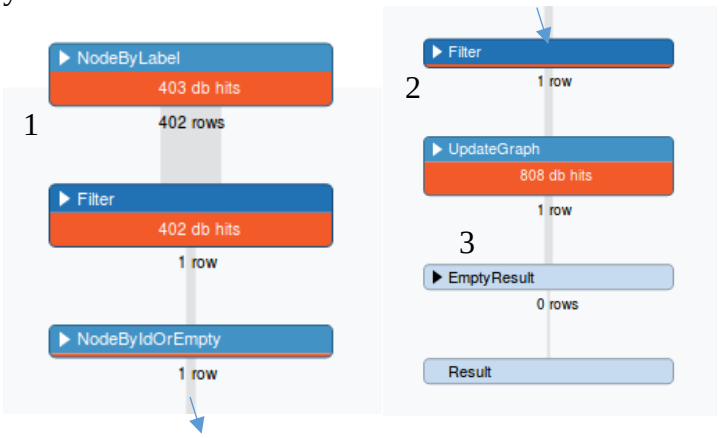
Asociar nodo de ADN a nodo de información

Usado en la opción de la aplicación Populate y Add DNA Sequence

```
MATCH (p1:DNASequence), (i1:Info) WHERE p1.nucleotides =~
"ATGGGCAAAGGAAGGCCAAATTTAA" AND ID(i1) = 3683
        CREATE UNIQUE (p1) -[:INFO]-> (i1),(i1) -[:INFO]-> (p1)
```

Tiempo de ejecución: 58 ms

Esta query busca todos los nodos de tipo ADN y todos los nodos de información y entre ellos elige aquellos que tengan unos nucleótidos en los de ADN y un id en el caso de los nodos de información específicos. Finalmente crea una doble asociación entre el nodo de información y el nodo de ADN.



Esquema 8 Asociar nodo de ADN a nodo de información

Como ya he dicho se eligen todos los nodos (1), y se realiza el producto cartesiano entre todos los nodos elegidos, lo cual en este caso no será visible puesto que la base de datos sólo tiene un nodo de información. Posteriormente se ve cómo ocurren los distintos filtros, reduciendose a una fila de datos (2), para finalmente crear la relación (3).

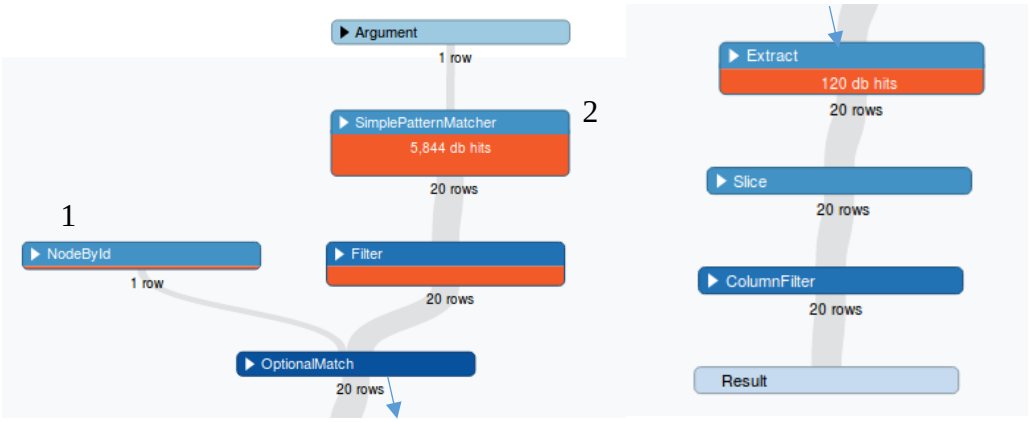
Obtiene información sobre un nodo de información

Usado en la opción de la aplicación *Get Info*

```
START i1= node(3683)  MATCH(i1) WITH i1
        OPTIONAL    MATCH    (i1)-[r:INFO]->(p1:DNASequence)-[:TRANSLATES]-
>(a1:AminoacydSequence)-[:STRUCTURES]->(s1:Structure)
        return i1.name AS InfoName, i1.sciname AS InfoSciName, i1.startCodons
As InfoStartCodons, ID(i1) As InfoId,
        p1.nucleotides AS DNASequence, ID(p1) as DNAId, a1.aminoacyds as
AminoacydSequence, ID(a1) as AminoacydId,
        s1.structure as Structure , ID(s1) as StructureId LIMIT 20
```

Tiempo de ejecución: 162 ms

Esta petición primero extrae el nodo con una id específica y después busca su secuencia asociada, la secuencia de aminoácidos asociada a esta y la estructura secundaria de la cadena de aminoácidos, para luego devolver los datos de todo, aunque con el límite de 20 por eficiencia.



Esquema 9 Obtiene información sobre un nodo de información

Se puede ver cómo lo primero que se hace es encontrar el nodo del que buscamos la ID, para, por otro lado buscar los nodos y relaciones que tienen las relaciones especificadas encima. De los nodos encontrados sólo se queda con 20 por el límite. Tiene sentido el bajo tiempo de ejecución teniendo en cuenta el hecho de que se trabaja con muy pocas filas.

Obtener información del ADN

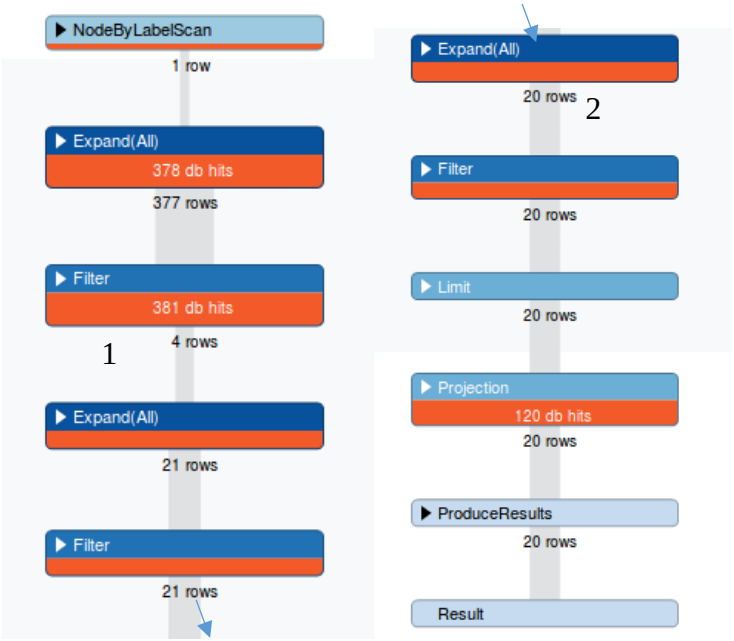
Usado en la opción de la aplicación Get Info DNA.

```
MATCH (i1:Info)-[r:INFO]->(p1:DNASequence)-[:TRANSLATES]->
        (a1:AminoacydSequence)-[:STRUCTURES]->(s1:Structure)
    WHERE p1.nucleotides =~ ".*"
    return i1.name AS InfoName, i1.sciname AS InfoSciName, i1.startCodons As
InfoStartCodons, ID(i1) As InfoId,
    p1.nucleotides AS DNASequence, ID(p1) as DNAId, a1.aminoacyds as
AminoacydSequence, ID(a1) as AminoacydId,
    s1.structure as Structure , ID(s1) as StructureId LIMIT 20
```

Tiempo de ejecución: 71ms

Esta operación busca encontrar 20 de los nodos de ADN cuyos nucleótidos cumplen la expresión regular ".*" y está asociado con un nodo de información, un nodo de proteínas y un nodo de estructura. Finalmente se devuelven como máximo datos de 20 filas. Entre estos datos se han devuelto los distintos ID(s) y los distintos campos de texto de cada uno de los nodos

(*nucleotides*, *aminoacyds* y *structures*) y además los codones de inicio, y un nombre real y un nombre científico para etiquetar los datos.



Esquema 10 Obtener información del ADN

Se ve cómo al ser el primer filtro sobre un nodo se reduce un poco el número de datos en (1), para después realizar el filtro y expandir las distintas operaciones (2), para finalmente devolver los datos de los nodos. Como en ningún momento se tratan una gran cantidad de datos el tiempo de ejecución es muy razonable.

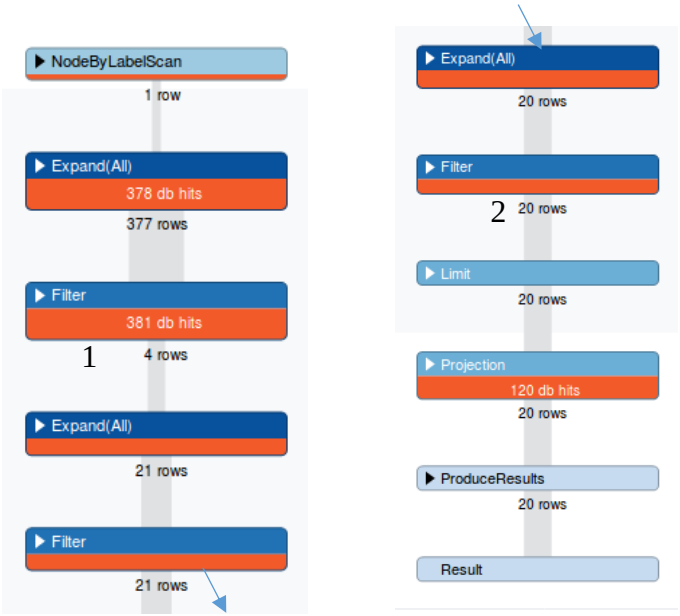
Obtener información de las proteínas

Usado en la opción de la aplicación *Get Info Protein*

```
MATCH(i1:Info)-[r:INFO]->(p1:DNASequence)-[:TRANSLATES]
    ->(a1:AminoacydSequence)-[:STRUCTURES]->(s1:Structure)
  WHERE a1.aminoacyds =~ "ATGG.*"
  return i1.name AS InfoName, i1.sciname AS InfoSciName, i1.startCodons As
InfoStartCodons, ID(i1) As InfoId,
   p1.nucleotides AS DNASequence, ID(p1) as DNAId, a1.aminoacyds as
AminoacydSequence, ID(a1) as AminoacydId,
   s1.structure as Structure , ID(s1) as StructureId LIMIT 20
```

Tiempo de ejecución: 71 ms

Esta operación es muy parecida a la anterior, y al igual que esta, busca encontrar 20 de los nodos de aminoácidos cuyos nucleótidos cumplen la expresión regular ".*" y está asociado con un nodo de información, un nodo de proteínas y un nodo de estructura. Finalmente se devuelven como máximo datos de 20 filas. Entre estos datos se han devuelto los distintos ID(s) y los distintos campos de texto de cada uno de los nodos (*nucleotides*, *aminoacyds* y *structures*) y además los codones de inicio, y un nombre real y un nombre científico para etiquetar los datos.



Esquema 11 Obtener información de las proteínas

Se ve cómo al ser el primer filtro sobre un nodo se reduce un poco el número de datos en (1), para después realizar el filtro y expandir las distintas operaciones (2), para finalmente devolver los datos de los nodos. Igualmente, en ningún momento hay un gran número de datos que tratar, por lo tanto el tiempo de ejecución es muy reducido.

Obtener todos los nodos de información

Usado en la opción de la aplicación Get All Info

```
MATCH (i1: Info)
    return i1.name AS InfoName, i1.sciname AS InfoSciName, i1.startCodons As
InfoStartCodons, ID(i1) as InfoId
```

Tiempo de ejecución: 91 ms

Esta petición busca y devuelve todos los nodos de información en forma de filas con los datos. Es una operación simple que no requiere muchos recursos y tan sólo busca las filas y las devuelve.

Crear nodo de información

Usado en Add Info

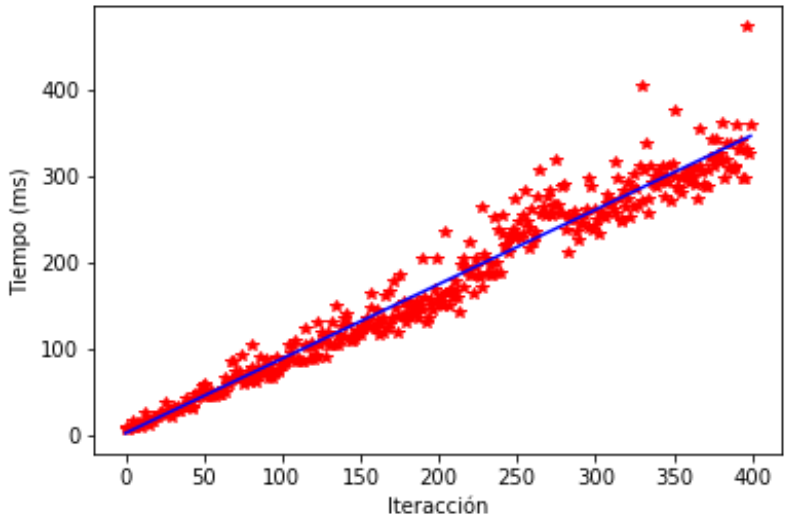
```
CREATE (i1:Info{startCodons:['ATG'] , name:'LOL' , sciname:'LAL'
}) return ID(i1) as Id
```

Tiempo de ejecución: 127 ms
Esta petición ejecuta la creación de un nodo de tipo Info con los distintos atributos especificados
Es una operación simple al igual que la anterior y que no requiere muchos recursos, pero al comparar con la anterior, detectamos que es más pesado el introducir datos que el buscarlos.

5.2 Pruebas sobre la aplicación

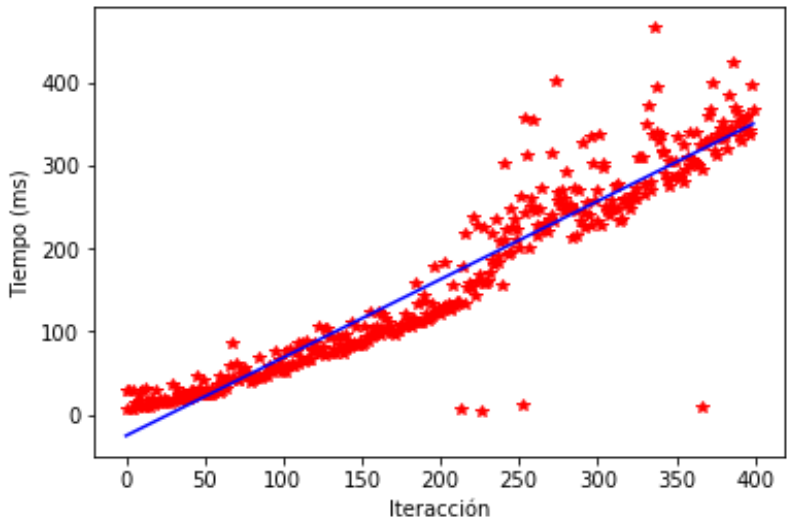
Hemos realizado las pruebas sobre las inserciones realizando la ejecución de un script con 400 inserciones de ADN que además ejecuta para cada una de estas inserciones el alineamiento del

ADN, la traducción, el alineamiento de proteínas y la predicción de la estructura secundaria. Hemos medido la ejecución de cada una de estas operaciones



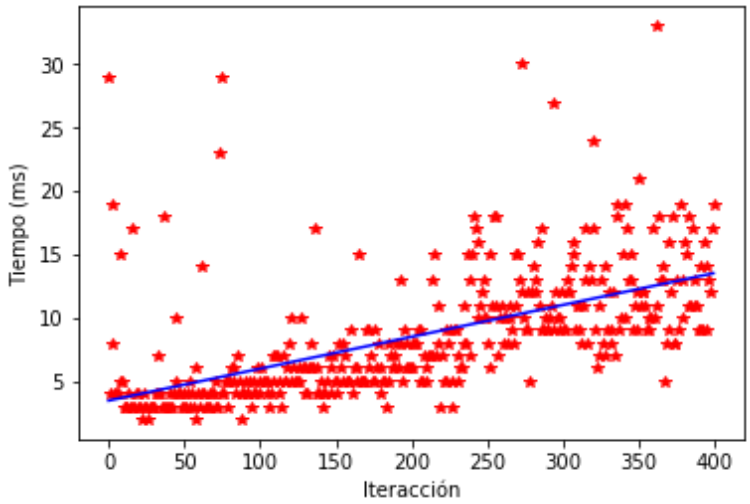
Grafica 1 Ejecución de un script que crea el nodo de ADN y llama al procedimiento swexec con el nodo creado que alinea la secuencia de nucleótidos comparado con una función O(n)

Como se puede ver en la gráfica previa, la ejecución de este query es O(n) siendo n el número de nodos de tipo ADN en el grafo. Tiene sentido que esto sea así puesto que la ejecución alinea el nodo objetivo con todos los demás por lo tanto para cada uno de los nuevos nodos realiza swexec con cada uno de los antiguos, lo cual acaba siendo O(n).



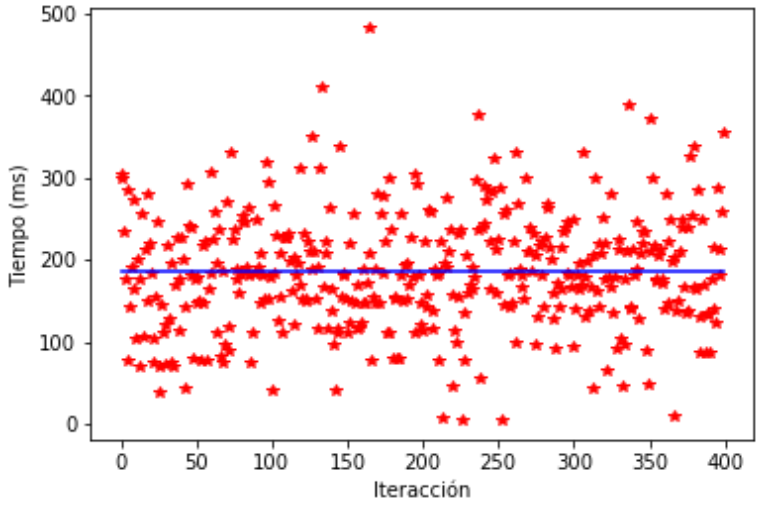
Grafica 2 3 Ejecución de un query que llama al procedimiento swexec que alinea las secuencias de aminoácidos que no tengan relaciones de tipo ALLIGN con las demás secuencias de la base de datos comparado con una función O(n).

Este caso es muy parecido al anterior, pero aquí se ve cómo las ejecuciones no son O(n) exactamente y la línea difiere un poco. Sabemos que, puesto que se parece a O(n) depende de alguna manera de los nodos de ADN, pero no de una manera tan directa como el anterior caso. Esto es, en este caso, al haberse realizado la traducción se han generado un número indeterminado de nodos de proteínas, como ya hemos explicado en el apartado 4.2.



Grafica 4 Ejecución de un query que llama al procedimiento proteinTranslate que traduce una cadena de ADN a una cadena de aminoácidos y crea un nodo con esta última (si no existe) y una relación entre la cadena de ADN y la de aminoácidos comparado con una función O(n)

Se puede ver cómo esta ejecución es bastante estable, a pesar de que aumenta el tiempo de ejecución dependiendo del tiempo un poco, pero no es nada demasiado importante comparado con las demás ejecuciones, esto es porque no tiene operaciones tan relacionadas con la comparación de nodos, como tienen las dos gráficas anteriores. En este caso sólo depende del número de nodos a la hora de buscar algunos datos a la hora de poblar la base de datos y buscar repetidos.



Grafica 5 Ejecución de un query que llama al procedimiento proteinStructure que realiza la predicción de la estructura secundaria de las proteínas comparado con una función O(1)

Finalmente se ve esta gráfica, la cual se puede observar que tiene una forma bastante dispar, esto es puesto que el mayor gasto de tiempo se produce dentro de la extensión al ejecutar el algoritmo de predicción de la estructura. Tiene sentido este tiempo puesto que es un algoritmo en Python, por lo tanto es mucho más costosa la ejecución que en Java.

5.3 Conclusiones

Como hemos visto y explicado, la eficiencia tanto de los distintos procedimientos como de las peticiones que se prevé que se van a realizar a la base de datos depende del número de datos que tendrá esta y, por supuesto, del sistema que aloje la base de datos, especialmente las operaciones que pueblan la base de datos.

Dentro de las operaciones que pueblan la base de datos, se puede ver que aquellas que son más costosas son las de alineamiento, que trabajan con todo el grueso de las relaciones, aun así, no se espera que este tipo de operaciones se usen mucho en la base de datos, si no que las operaciones más usadas sean las de búsquedas.

Dentro de las operaciones de búsqueda, estudiando las distintas peticiones, se puede ver que la eficiencia de las peticiones depende del número de nodos con los que va a trabajar la petición, por lo tanto se puede conseguir una relativa eficiencia sugestionando al usuario de la aplicación de alguna manera para que realice peticiones que trabaje con conjuntos acotados de datos, en vez de con todos los datos de la base de datos.

7 Conclusiones y trabajos futuros

7.1 Conclusiones

Este Trabajo de Fin de Grado tenía como objetivo crear un ecosistema para trabajar con distintos datos de tipo biológico, a continuación vamos a explicar las distintas conclusiones a las que hemos llegado después de hacerlo.

Como ya se ha dicho durante todo el trabajo, se decidió usar la base de datos basada en grafos, Neo4j, como base, sobre la cual hemos creado varias extensiones que realizan las operaciones listadas en los objetivos: la alineación genética, para lo cual hemos usado el algoritmo de Smith-Waterman, la síntesis de proteínas, para lo cual hemos creado nuestro propio algoritmo de síntesis a partir de la tabla de codones con su respectiva traducción y, finalmente, la predicción de la estructura secundaria de las proteínas, para la cual se ha usado el algoritmo probabilístico basado en Chou-Fassman. Estas implementaciones y el diseño de la base de datos implican que se ha cumplido el primero de los objetivos expuestos al principio de la memoria.

El segundo de los objetivos consistía en estudiar la eficiencia de las posibles operaciones que se pueden ejecutar contra la base de datos. Como hemos podido ver en el apartado de pruebas, el alineamiento de nuevos datos dependerá del número de datos, y por lo tanto, cuantos más datos haya en la base de datos, más tardara. La búsqueda sobre la base de datos también depende del número de datos, y es cierto que cuando se buscan muchos datos es posible que tarde demasiado, pero si se filtran los nodos antes de realizar búsquedas sobre relaciones, las peticiones resultantes son muy rápidas, por lo tanto la eficiencia de las búsquedas depende de la gestión que se realice en la aplicación sobre los nodos sobre los que se realizará la búsqueda. Esto implica que si se realiza una buena gestión las operaciones contra la base de datos se mantendrán a con un coste temporal aceptable.

Finalmente, hemos realizado una aplicación de prueba de este ecosistema, en la que se ve un ejemplo de cómo se podría usar este proyecto. En esta aplicación se han implementado todas las posibles operaciones básicas que tenían sentido en el contexto del problema:

7.2 Trabajo futuro

Existen diversas formas con las que se podría continuar el trabajo.

En primer lugar, se pueden crear otros procedimientos para la base de datos, especialmente para el estudio de las estructuras terciaria y cuaternaria de las proteínas, que son los procesos más significativos después de los ya realizados, y otros tipos de distintos estudios sobre los datos guardados en la base de datos, como podría ser un detector de mutaciones que han ocurrido ya, o un predictor de la posibilidad de que ocurran mutaciones por la creación de bucles en la cadena de ADN durante la traducción de la cadena a proteínas.

Igualmente se pueden crear más aplicaciones para el uso de las diversas extensiones que se adecuen a la necesidad y los requerimientos de los investigadores que vayan a usar la aplicación, especialmente consideramos que podría ser interesante crear una aplicación que implemente distintos roles de usuario, y por lo tanto permita que haya usuarios que sólo realicen búsquedas, y además otros usuarios que pueden añadir cadenas.

Glosario

- **Ácido desoxirribonucleico (ADN)**: El **ácido desoxirribonucleico** es una macromolécula que codifica los genes de las células, bacterias y algunos virus. Esta información genética del ADN se usa para fabricar las proteínas necesarias para el desarrollo y funcionamiento del organismo.
- **Alineamiento**: Un **alineamiento de secuencias** es una forma de representar y comparar dos o más secuencias o cadenas de ADN, ARN, o estructuras primarias proteicas para resaltar sus zonas de similitud, que podrían indicar relaciones funcionales o evolutivas entre los genes o proteínas consultados.
- **Aminoácido**: Los aminoácidos son las unidades químicas o elementos constitutivos de las proteínas.
- **Bioinformática**: La **bioinformática** es la aplicación de tecnologías computacionales a la gestión y análisis de datos biológicos.
- **Codón:** Grupo de tres nucleótidos en el ADN
- **Genoma**: Conjunto de genes y disposición de los mismos en la célula
- **Información genética, material genético:** Guarda la información genética de una forma de vida orgánica.
- **Nucleótido**: Compuesto químico orgánico fundamental de los ácidos nucleicos, constituido por una base nitrogenada, un azúcar y una molécula de ácido fosfórico.
- **Péptido**: Los péptidos son un tipo de moléculas formadas por la unión de varios aminoácidos mediante enlaces peptídicos
- **Polipéptido**: Son péptidos con muchos aminoácidos.
- **Proteína**: Sustancia química que forma parte de la estructura de las membranas celulares y es el constituyente esencial de las células vivas.
- **Síntesis proteica** : Es el proceso anabólico mediante el cual se forman las proteínas

Bibliografía

[1] «GenBank,» [En línea]. Available: https://www.ncbi.nlm.nih.gov/genbank/. [Último acceso: 24 06 2017].

[2] «European Nucleotide Archive,» [En línea]. Available: http://www.ebi.ac.uk/ena. [Último acceso: 24 06 2017].

[3] «UniProt,» [En línea]. Available: http://www.uniprot.org/. [Último acceso: 22 06 2017].

[4] «Protein Data Bank,» [En línea]. Available: http://www.rcsb.org/pdb/home/home.do.

[5] A. Jimeno, M. Ballesteros, L. Ugedo y M. A. Madrid, *Biología 2 Bachillerato*, Santillana, 2009.

[6] R. R. Sinden, *DNA structure and function*, Elsevier, 2012.

[7] J. Slonczewski y J. W. Foster, *Microbiology: An Evolving Science*, New York: W.W. Norton & Co, 2009.

[8] J. A. Mina Caicedo y F. J. Romero-Campero, «Biological Sequence Alignment,» [En línea]. Available: https://www.cs.us.es/~fran/students/julian/sequence_alignment/sequence_alignment.html. [Último acceso: 21 04 2017].

[9] R. Deonier, S. Tavaré y M. Waterman, *Computational Genome Analysis: an introduction.*, Springer-Verlag, 2005.

[10] G. T. M. Erik P. Garrison, «SSW Library: An SIMD Smith-Waterman C/C++ Library for Use in Genomic Applications,» 4 12 2013. [En línea]. Available: http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0082138. [Último acceso: 22 04 2017].

[11] S. Parthasarathy, «Sequence Alignment Algorithms - Application to Bioinformatics Tool Development,» Bharathidasan University, Tiruchirappalli.

[12] R. Salvador Gradaille, «Aceleración del algoritmo Smith-Waterman,» Trabajo de Fin de Master, Escuela Politécnica Superior, Universidad Autónoma de Madrid,, 2010/2011.

[13] V. O. Polyanovsky, M. A. Roytberg y V. G. Tumanyan, «Comparative analysis of the quality of a global algorithm and a local algorithm for alignment of two sequences,» *Algorithms for Molecular Biology*, 2011.

[14] H. Carrillo y D. Lipman, "The Multiple Sequence Alignment Problem in Biology", *SIAM Journal of Applied Mathematics*, 1988.

[15] D. Mount, *Bioinformatics: Sequence and Genome Analysis (2nd ed.).*, NY: Cold Spring Harbor Laboratory Press: Cold Spring Harbor, 2004.

[16] NCBI, «Blast,» [En línea]. Available: https://blast.ncbi.nlm.nih.gov/Blast.cgi. [Último acceso: 24 06 2017].

[17] T. F. Smith y M. Waterman, «Identification of common molecular subsequences,» *Journal of Molecular Biology*, nº 147, pp. 195 -197, 1981.

[18] M. S. Farrar, «Striped Smith–Waterman speeds database searches six times over other SIMD implementations,» *Bioinformatics.*, vol. 23, pp. 156-161, 2007.

[19] D. S. Hirschberg, «A linear space algorithm for computing maximal common subsequences,» *Communications of the ACM*, vol. 18, pp. 341-343, 1975.

[20] M. Waterman, T. Smith y W. Beyer, «Some biological sequence metrics,» *Advances in Mathematics,* vol. 20, p. 367–387, 1976.

[21] R. C. Lewontin, «The genetic basis of evolutionary change.,» *New York: Columbia University Press,* vol. Vol. 560, 1974.

[22] J. Kimball, *The Genetic Code,* Kimball's Biology Pages, 2014.

[23] A. Lehninger, *Principles of Biochemistry, 2nd Ed*, Worth Publishers, 1993.

[24] C. Branden y J. Author, *Introduction to protein structure (2nd ed.)*, NY: Garland Science, 1999.

[25] P. Y. Chou y G. D. Fasman, «Empirical predictions of protein conformation,» *Annual review of biochemistry 47.1*, vol. 47.1, pp. 251-276, 1978.

[26] P. Argos, M. Hanei y R. M. Garavito, «The Chou-Fasman secondary structure prediction method with an extended data base.,» *FEBS letters* , vol. 93.1, pp. 19-24, 1978.

[27] M. Charton, Charton y B. I., «The dependence of the Chou-Fasman parameters on amino acid side chain structure.,» *Journal of theoretical biology* , vol. 102.1, pp. 121-134, 1983.

[28] A. Rodríguez Sosa, «Uso de algoritmos ab initio para la predicción del plegamiento secundario de proteínas,» Trabajo de Fin de Grado, Facultad de Biología, Universidad Autónoma de Madrid,, 2015.

[29] E. Redmond y J. R. Wilson, *Seven Databases in Seven weeks*, Pragmatic Bookshelf, 2012.

[30] M. Stonebraker, «SQL databases v. NoSQL databases,» *Communications of the ACM 53.4,* pp. 10-11, 2010.

[31] Neo4j, «From Relational to Neo4j,» Neo Inc, [En línea]. Available: https://neo4j.com/developer/graph-db-vs-rdbms/. [Último acceso: 20 04 2017].

[32] O. Tezer, «A Comparison Of NoSQL Database Management Systems And Models,» [En línea]. Available: https://www.digitalocean.com/community/tutorials/a-comparison-of-nosql-database-management-systems-and-models. [Último acceso: 20 04 2017].

[33] D. Sullivan, «Types of NoSQL databases and key criteria for choosing them,» techtarget, [En línea]. Available: http://searchdatamanagement.techtarget.com/feature/Key-criteria-for-choosing-different-types-of-NoSQL-databases. [Último acceso: 2017 04 20].

[34] D. Sullivan, *NoSQL for Mere Mortals*, Pearson, 2015.

[35] M. Nicola, «5 Reasons for Storing XML in a Database,» *Native XML Database,* 28 Septiembre 2010.

[36] Neo4j, «How Graph Databases Relate To Other NoSQL Data Models,» [En línea]. Available: https://neo4j.com/developer/graph-db-vs-nosql/. [Último acceso: 20 04 2017].

[37] J. W. &. E. E. Ian Robinson, *Graph Databases New Opportunities For Connected* Data, O'Reilly, 2015.

[38] Neo4j, «Why Graph Databases?» [En línea]. Available: https://neo4j.com/why-graph-databases/. [Último acceso: 2017 04 21].

[39] Stackoverflow community, «Data mining with Neo4j» Stackoverflow, [En línea]. Available: http://stackoverflow.com/questions/22149288/data-mining-with-neo4j. [Último acceso: 21 04 2017].

[40] E. Genesky, «16 Graph Databases Compared» [En línea]. Available: https://dzone.com/articles/16-graph-databases-compared. [Último acceso: 20 04 2017].

[41] C. Kemper, *Beginning Neo4j*, Apress, 2015.

[42] R. V. Bruggen, *Learning Neo4j*, Packt Publishing, 2014.

[43] Neo4j, *Neo4j Cypher Refcard 3.2*, Neo Technology.

[44] Neo4j, «Procedures» Neo Technology, Inc., [En línea]. Available: http://neo4j.com/docs/developer-manual/current/extending-neo4j/procedures/. [Último acceso: 21 04 2017].

[45] Neo4j, «Neo4j Procedure API Reference» [En línea]. Available: https://neo4j.com/docs/java-reference/3.2/javadocs/index.html?org/neo4j/procedure/Procedure.html. [Último acceso: 28 04 2017].

[46] Jython, «Jython» [En línea]. Available: http://www.jython.org. [Último acceso: 24 04 2017].

[47] Yahoo, «PureCSS» [En línea]. Available: https://purecss.io/.

[48] A. Jacomy, «Sigma JS» 21 06 2017. [En línea]. Available: http://sigmajs.org/. [Último acceso: 21 06 2017].

[49] Node.js Foundation, «Node.js» Node.js Foundation, [En línea]. Available: https://nodejs.org/en/. [Último acceso: 21 06 2017].

[50] StrongLoop, IBM, and other ExpressJS contributors, «ExpressJS» [En línea]. Available: http://expressjs.com/. [Último acceso: 21 06 2017].

[51] ExpressJS, «Using template engines with Express» [En línea]. Available: http://expressjs.com/en/guide/using-template-engines.html. [Último acceso: 25 05 2017].

[52] Y. Yeen, «Express file uploads with Multer» [En línea]. Available: https://scotch.io/tutorials/express-file-uploads-with-multer. [Último acceso: 25 05 2017].

Anexo I: Tutorial de la aplicación de prueba detallado

Aquí podemos ver todas las operaciones que se pueden realizar en la aplicación con salida correcta, aquellas operaciones incorrectas provocan que aparezca un mensaje indicando el error.

Search Score DNA: Permite buscar todas las relaciones con un score mínimo entre los nodos que siguen un patrón de ADN. La salida en un grafo de tipo alineamiento.

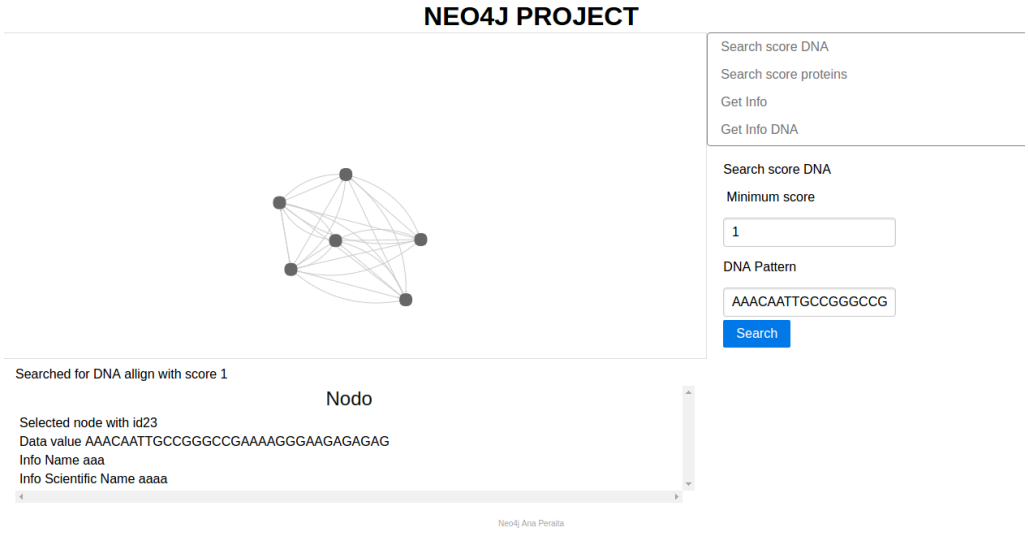


Ilustración 7 Aplicación de prueba, vista de búsqueda de ADN alineado.

Se ve un grafo de tipo alineamiento y debajo aparece información sobre un nodo al que se le ha hecho hover.

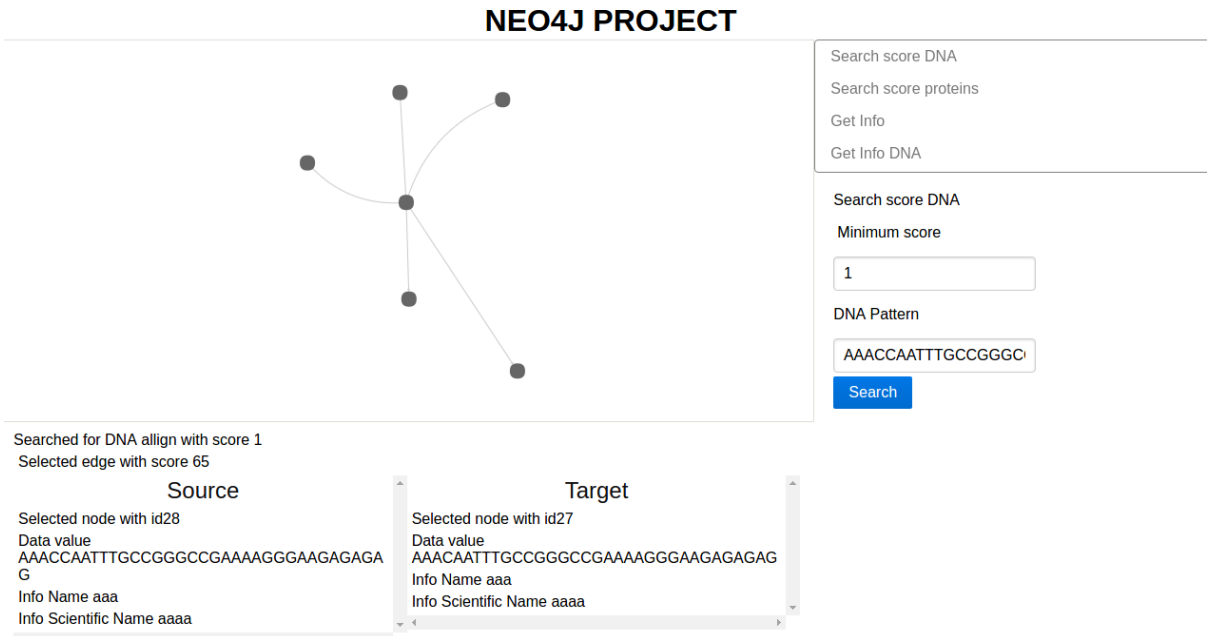


Ilustración 8 Aplicación de prueba, vista de búsqueda de ADN alineado.

Se ve un grafo de tipo alineamiento y debajo aparece información sobre una arista a la que se le ha hecho hover.

Search Score proteins: Muy parecido al anterior, sólo que busca en los nodos de proteínas.

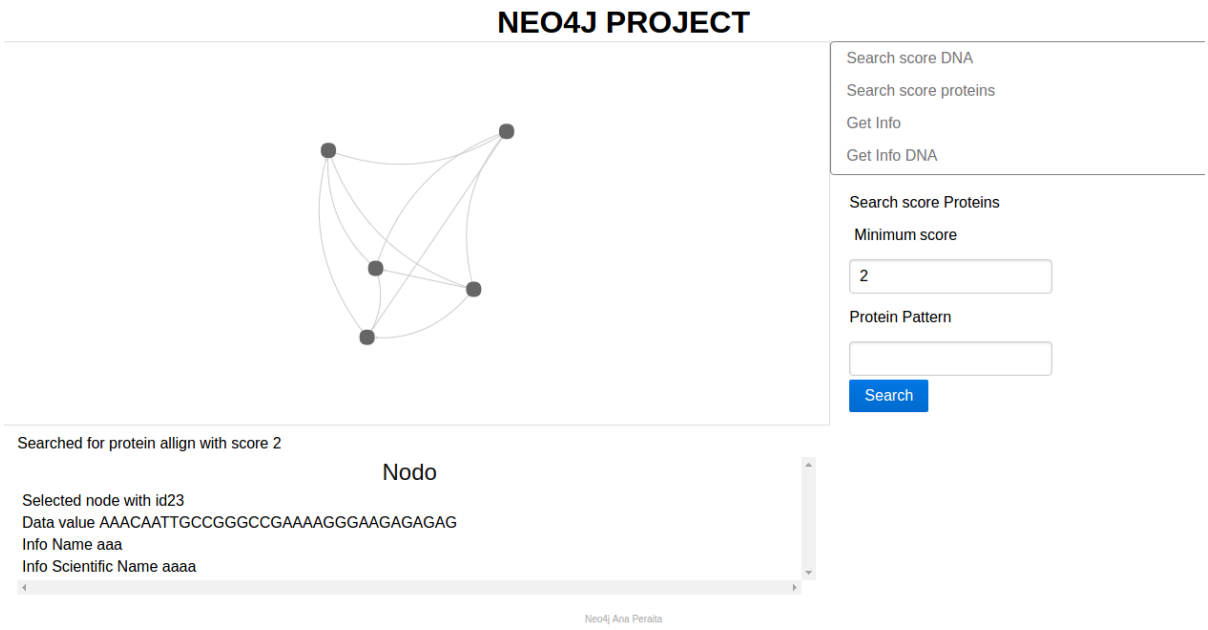


Ilustración 9 Aplicación de prueba, vista de búsqueda de aminoácidos alineados.

Se ve un grafo de tipo alineamiento y debajo aparece información sobre un nodo al que se le ha hecho hover.

Get Info: Devuelve información en forma de un grafo de información, con toda la información relacionada con un nodo de información que se busca por id.

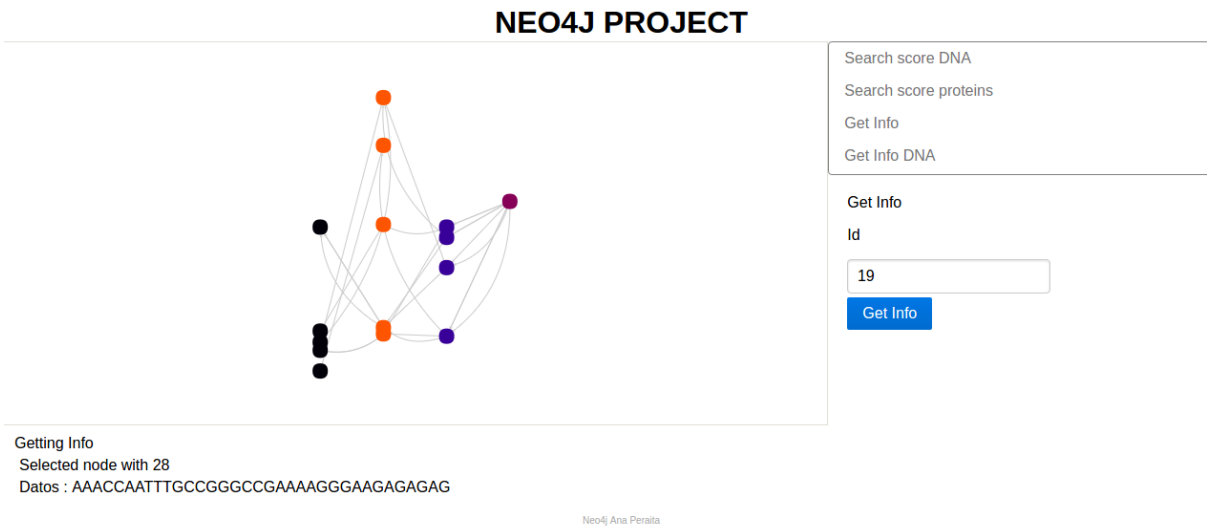


Ilustración 10 Aplicación de prueba, vista de búsqueda información por id.

Aquí podemos ver un grafo de información con el nodo buscado y todos los nodos de otros tipos asociados a este (ADN, proteínas y estructura).). Al hacer hover sobre los distintos nodos aparecen los datos asignados a los nodos.

Get Info DNA: Parecido al anterior pero busca sobre un patrón de nucleótidos de ADN.

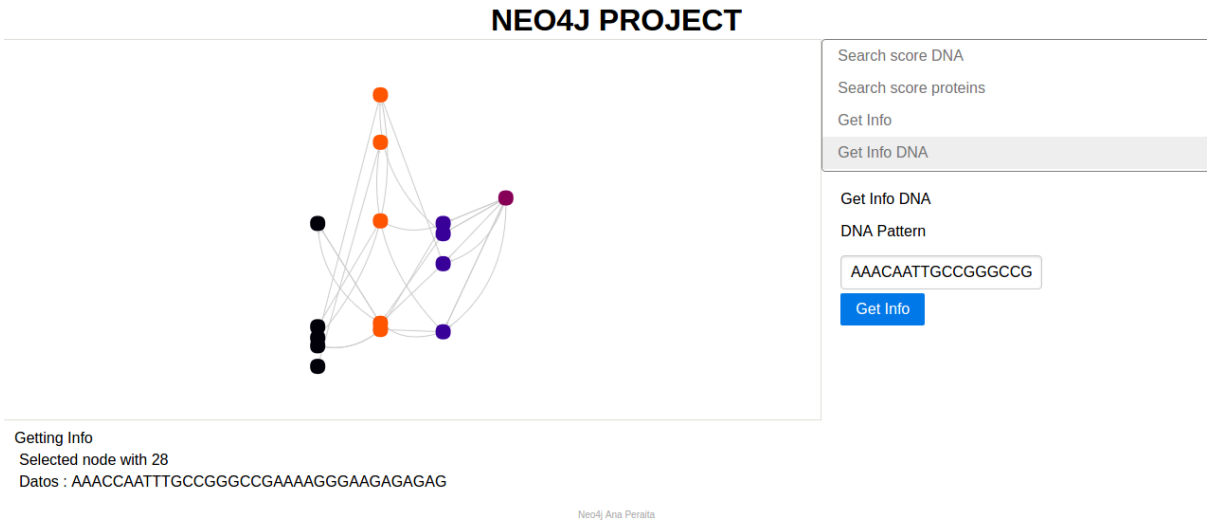


Ilustración 11 Aplicación de prueba, vista de búsqueda información por patrón de ADN.

Aquí podemos ver un grafo de información con los nodos buscados y todos los nodos de otros tipos asociados a este (Info, proteínas y estructura)). Al hacer hover sobre los distintos nodos aparecen los datos asignados a los nodos.

Get Info Protein: Parecido al anterior pero busca sobre un patrón de nucleótidos de proteínas.

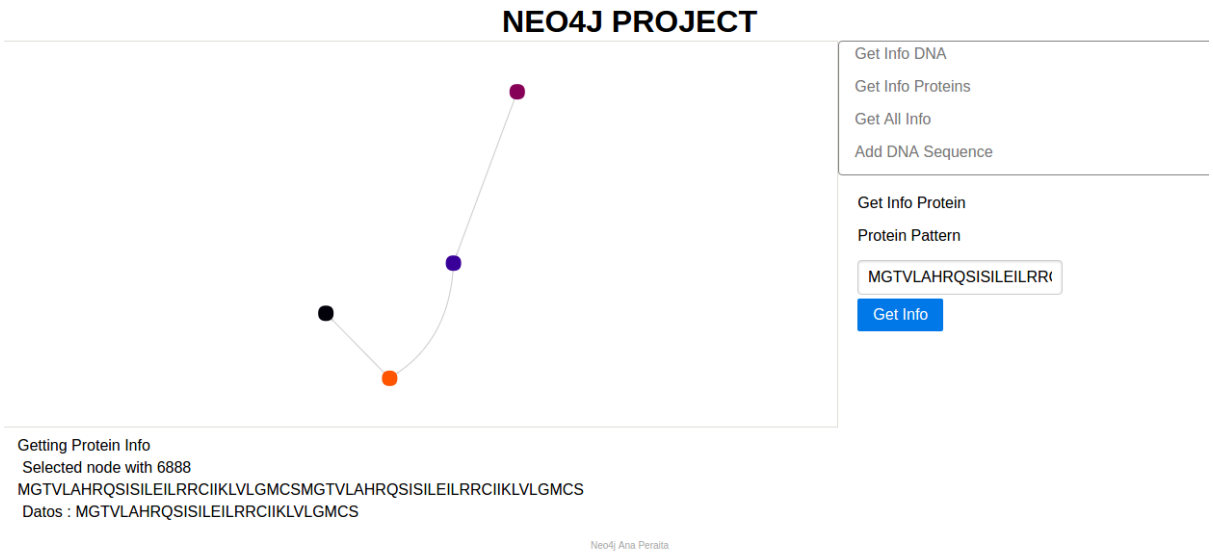


Ilustración 12 Aplicación de prueba, vista de búsqueda información por patrón de aminoácidos.

Aquí podemos ver un grafo de información con los nodos buscados y todos los nodos de otros tipos asociados a este (Info, ADN y estructura). Al hacer hover sobre los distintos nodos aparecen los datos asignados a los nodos.

Get All Info: Devuelve todos los nodos de información como un nodo de información.

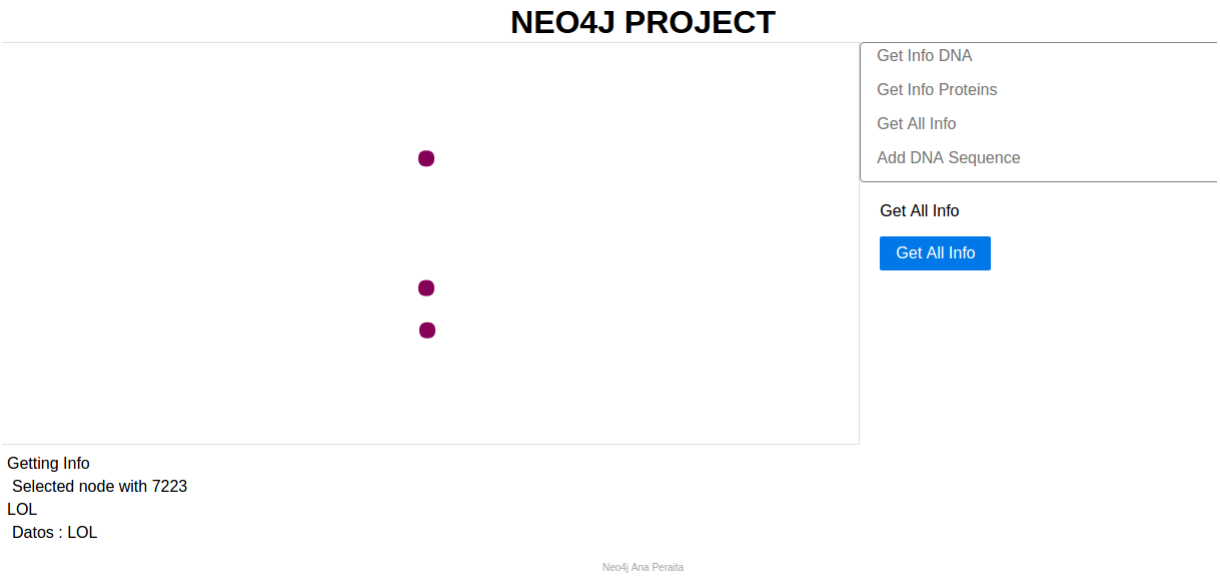


Ilustración 13 Aplicación de prueba, vista de muestra de nodos de información.

Aquí podemos ver un grafo de información con todos los nodos de tipo info

Add DNA Sequence: Añade una secuencia a la base de datos con un nodo de información (en forma de id) asociado.

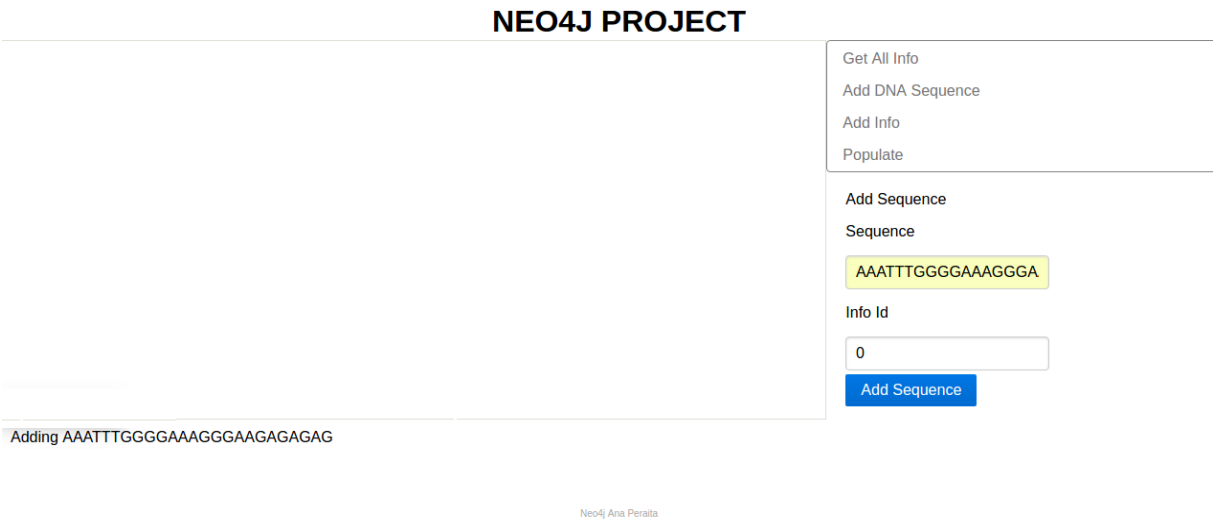


Ilustración 14 Aplicación de prueba, vista de añadir secuencia.

Aquí podemos ver una petición para añadir una secuencia y el mensaje de estado

Add Info: Añade un nodo de información con un nombre, un nombre científico y los codones de inicio.

Ilustración 15 Aplicación de prueba, vista de añadir información

Podemos ver el menú de la opción de Add Info y el mensaje de estado.

Populate: Añade varios nodos desde un fichero y lo asocia con un nodo de información en forma de id.

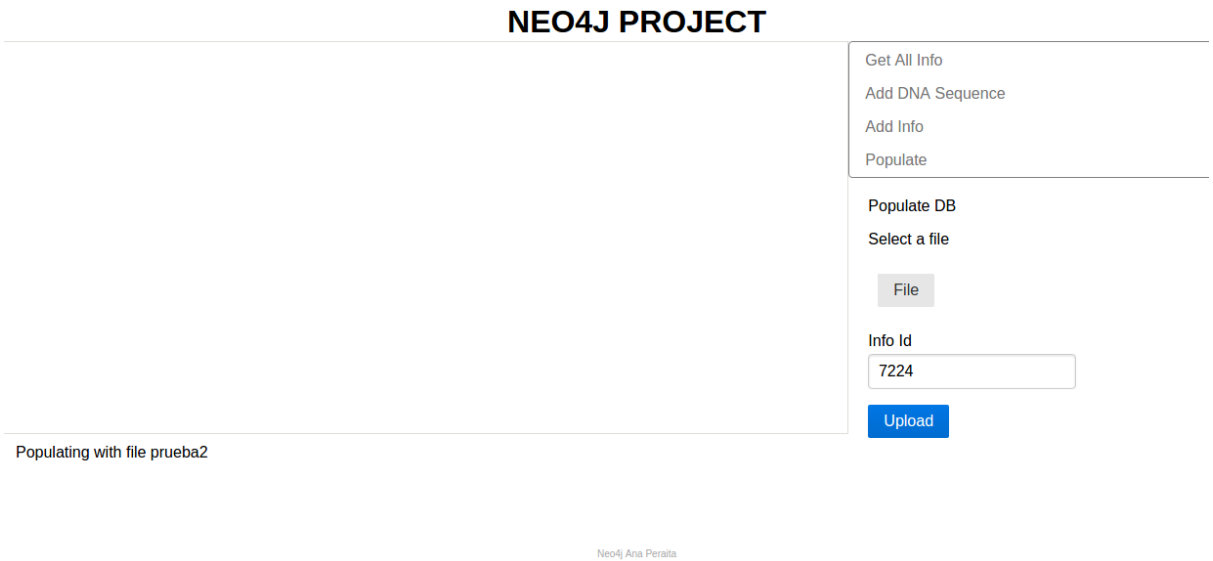


Ilustración 16 Aplicación de prueba, vista de populate

Podemos ver el menú de populate y el estado de la petición.